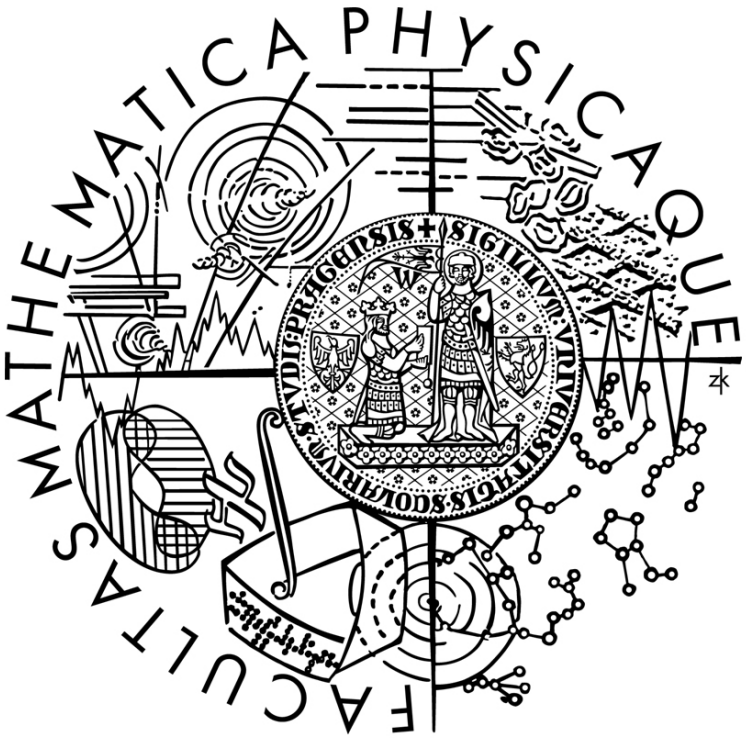
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Ondřej Palkovský

Implementace Time Addressable Storage

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Filip Zavoral, Ph.D.

Studijní program: Informatika, Softwarové systémy

2008

I would like to express my thanks to my supervisor, RNDr. Filip Zavoral, Ph.D. for patience and valuable suggestions that made the structure of this thesis better suit its purpose.

I thank my colleagues and friends from the company GAPP System for helping me set up the testing environment, namely to Petr Dvořák, David Gottwald and Petr Faltýn.

I sincerely thank Zuzana Šenkýřová and Simona Hopfingerová for reading and patiently correcting my linguistic mistakes in a very technical text.

I thank heartfully both my parents for encouragement in the time of deepest despair while burning midnight oil, when the bugs in my code evaded every effort to be discovered.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 25. 7. 2008 Ondřej Palkovský

Contents

List of Figures

List of Tables

Název práce: Implementace Time Addressable Storage
Autor: Ondřej Palkovský
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Filip Zavoral, Ph.D.
E-mail vedoucího: Filip.Zavoral@mff.cuni.cz
Abstrakt: *Přehled technologií používaných k udržování historických verzí dat — zálohování, snapshoty, asynchronní replikace, HSM, Storage Area Network, virtualizace úložných prostor. Popis vrstvy device mapper implementvané v současných verzích Linuxu, popis funkcionality současného systému řízení snapshotů včetně integrace v administrátorském rozhraní. Návrh a implementace nového systému snapshotů založeného na architektuře snapshot/view se snapshoty řetězenými za sebou. Návrh a implementace diskového žurnálu pro urychlení zápisových operací. Integrace celého řešení do administrátorského rozhraní LVM.*
Klíčová slova: Linux, snapshot, LVM, CDP

Title: Implementation of Time Addressable Storage
Author: Ondřej Palkovský
Department: Department of Software Engineering
Supervisor: RNDr. Filip Zavoral, Ph.D.
Supervisor's e-mail address: Filip.Zavoral@mff.cuni.cz
Abstract: *Overview of technologies used to keep historical versions of data — backups, snapshots, asynchronous replication, HSM, Storage Area Network, storage virtualization. Description of the programming layer device mapper implemented in current versions of Linux, description of the functionality of the current system of snapshots including integration in the userspace administrative tools. Design and implementation of a new snapshot system based on snapshot/view architecture with chained snapshots. Design and implementation of disk journal to speed up the write operations. Integration of the whole solution into LVM.*
Keywords: Linux, snapshot, LVM, CDP

Chapter 1

Introduction

1.1 Motivation

A great majority of daily business activities would immediately come to a halt if IT systems of the company ceased working. This typically includes not only IT companies – software companies, e-shops, but increasingly all companies in all sectors of the economy. As the majority of system failures is caused by user or software errors [5], the ability to access historical copies of data in order to resume accessibility of IT systems becomes a crucial part of Business Continuance strategy in most companies.

Business continuity solutions concerning storage systems are evaluated by assessing two major criteria: Recovery Point Objective (maximum amount of data allowed to be lost) and Recovery Time Objective (time needed to recover data). This thesis surveys several different techniques used to store historical copies of data and evaluates how they can be used to meet these criteria.

1.2 Goals

This thesis aims to develop a new snapshot driver that could be used as a means to provide backup copies of data. The driver would have reasonable impact on performance of the storage system while meeting stringent recovery time and recovery point objectives. In particular, the snapshot driver should reach the following aims:

- implementation of snapshot/view architecture
- implementation of disk journal to improve performance
- integration into existing Logical Volume Manager environment

This thesis will discuss different ways to implement these features with regard to performance and data consistency.

Chapter 2

Time Addressable Storage

Several solutions exist to ensure accessibility of storage systems. Most of the classical solutions are aimed at strengthening physical resilience and keeping redundant copies of data; ensuring that no hardware failure could render a storage system inaccessible. However, when a user error or software data corruption occurs, these systems blindly ensure that all the corrupted data is safely written on the disk. Therefore, demand exists for solutions that could deliver historical versions of data, so that an IT system can be rolled back to the state before the data corruption occurred. Such systems could be called 'time addressable storage'.

2.1 Backups

The problem of business continuity has traditionally being solved by copying periodically data to other media – by making backups. The most classical solution is a centralized backup server with network-connected clients. A central backup server manages the whole backup process and allows administrators to oversee the backup activities from one place.

Classical backup solutions allow administrators to define rules of how long the old backups should reside on the backup media before they are overwritten, according to company policy. In order for the backups to be useful for a full disaster recovery, some of the backup media should be transferred to a geographically different location, so that in case of disaster the backup media is not destroyed along with the rest of the IT infrastructure.

Current IT environment is very data intensive – the databases often comprise hundreds of gigabytes or terra-bytes of data. In order to minimize the backup window (the period of time when the system cannot be used because of making backups) and impact on production servers, the companies resort to making incremental and differential backups. The basis for all the backups is a 'full backup' – a full copy of the on-disk data. A *differential backup* consists of changes between current state and the last full backup. An *incremental backup* comprises of data that have been changed

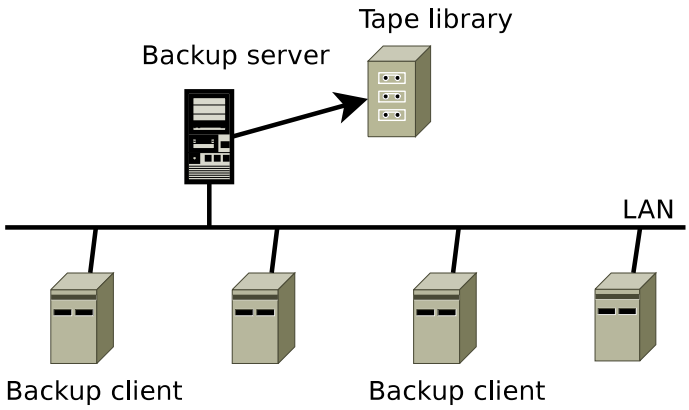


Figure 2.1: Centralized backup system

since last full or differential backup.

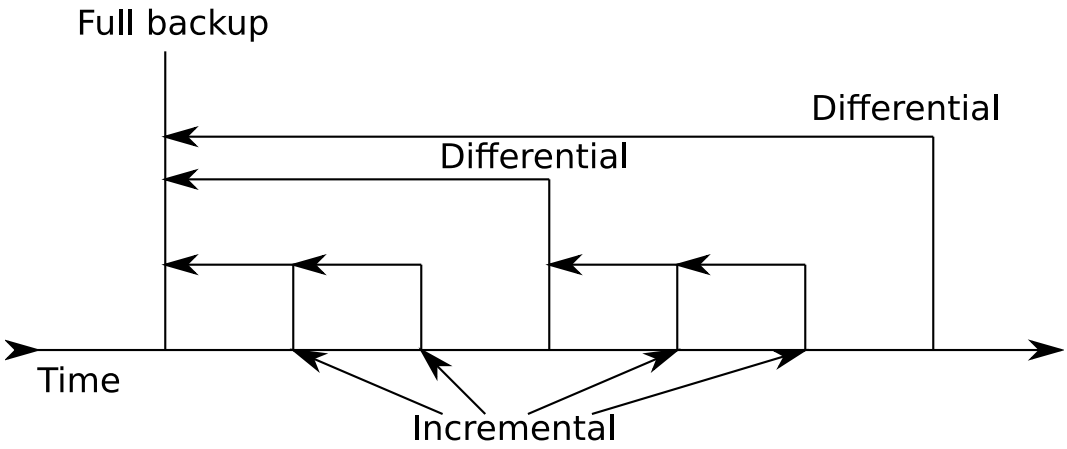


Figure 2.2: Incremental and differential backups

The use of incremental and differential backups enhances granularity of the data history and reduces the backup window. However, it also widens the recovery time because the data that should be recovered to a certain point in time must be assembled from different backups. The incremental and differential backups can take a block-level or a file-level form. The most common backup solutions use file-level incremental backups by backing up only the files that have been changed since the corresponding last incremental or full backup. Unfortunately, the incremental backups usually do not track deleted files – it may happen, that once the recovery is required, the recovery process tries to recover more data than can actually fit on the disk.

2.1.1 Database Backups and Recovery

The traditional backup systems fall into problems when backing up live data. A backup client reads data sequentially while the database system changes the same data randomly. The result is an inconsistent backup which is usually useless. The database systems are an example of such systems, where most of the data bears this characteristics.

The first database systems had to be backed up by using 'offline backups'. The database had to be shut down, then the backup was made and the database was started again. This method became insufficient when the time needed to backup the data got longer then the available backup window. In current 24/7 business environment the backup window is essentially nil and the offline backups are no more used to backup databases.

An alternative to offline backup is a database dump. The backup client uses standard database access to fetch data from the database. This technique can be used without shutting down the database but it is very slow and CPU-intensive. The requirements for the Recovery Point Objective to make at least one backup a day are rendered impossible considering the fact that the dump of the database can take more than 24 hours. The recovery of a database from a database dump is very slow as well.

The database vendors decided to solve this problem by allowing 'online backups'. When the database is set to 'backup mode', it logs all the changes to the database files into a separate change file. When the backup of the database file is complete, this change file is backed up as well. To recover the database, the database files are recovered first and then the changes, that were made during the backup are applied. This allows making backup without disrupting the business operations, however, there is usually some impact on throughput of the system during backup both because of the utilization of disk storage and because of the need of the database to track the changed data. For example the Oracle database stores this information in the redo-log files.

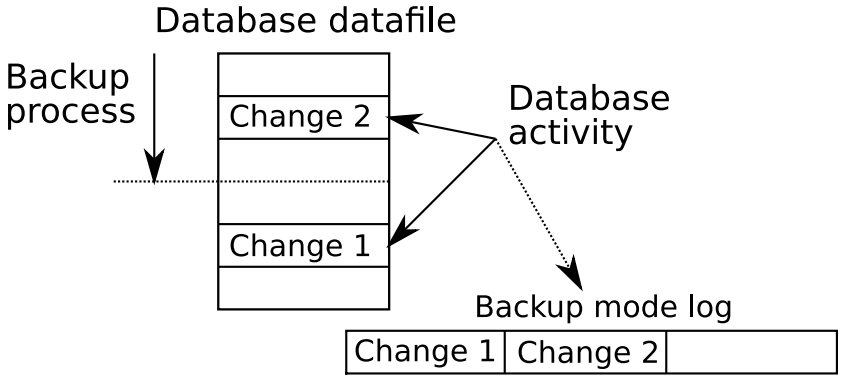


Figure 2.3: Online backup of database datafiles

In order to survive power failure and speed things up, most databases include so

called ‘redo-log’. It is a journal of operations that are to be performed on the database. When an operation is written into the redo-log, the client can be notified that requested operation is complete, while the database is updating the actual database tables in the background. If a power failure occurs, the database simply replays the acknowledged operations from the redo-log and the database datafiles are made consistent.

The redo-logs can be utilized as an easy way to make incremental backups. When a redo log area is filled, the database can ask the backup system to backup the redo-log. Upon the recovery, data from some full backup are restored along with subsequent redo-logs; these redo-logs are then applied to the database. The administrator is allowed to specify a point in time when the application of the redo-logs should stop. The redo-log backup and restore thus provides a transaction-level granularity of restore; very high level of security against user errors; and if the backup system is configured to backup the redo-logs automatically as soon as possible, a recovery point that loses almost no data.

Sometimes faster backup leads to drastically slower recovery time – creating database backups using redo logs is such case. In a heavily utilized system the application of the redo-logs can take as much time as was needed to change the data in the first place. If an organization makes e.g. one full backup per week, the application of several days of redo-logs can take several days as well. To address this problem the database vendors started providing modified backup clients that could read directly the database datafiles and extract only the blocks that had been changed since the last backup – essentially providing block-level incremental and differential backups. This access bypasses the database transaction system and is very fast and efficient.

By combining online, incremental and redo-log backups the administrators can achieve a very good level of recovery point and recovery time while minimizing the impact of the backup procedures on the normal business activity even in the 24/7 environment, while being able to recover the data fast to any point in time within certain historical time frame with transaction level granularity.

2.1.2 Split-mirror

Big enterprise disk storage is often used for the big database systems. These hardware systems often provide functionality that facilitates further reduction of backup procedure impact on ordinary business operation. For example EMC storage systems provide Business Copy Volume functionality, Hitachi provides TrueCopy function [12].

The basis lies in a classical volume mirror (RAID 1). When a backup is to be made, the database is set into the backup mode, the mirroring is stopped and the database is set back from backup mode. The backup software then starts the backup from the second disk from the pair, preferably from a different server. The disk system then maintains a map of changed blocks, so that when a request for reestablishing the mirror arises, only the changed blocks are mirrored;

The disks used for backup can be in a disconnected state most of the time and serve as a hot backup that is readily prepared to be used in case of disaster, which can be

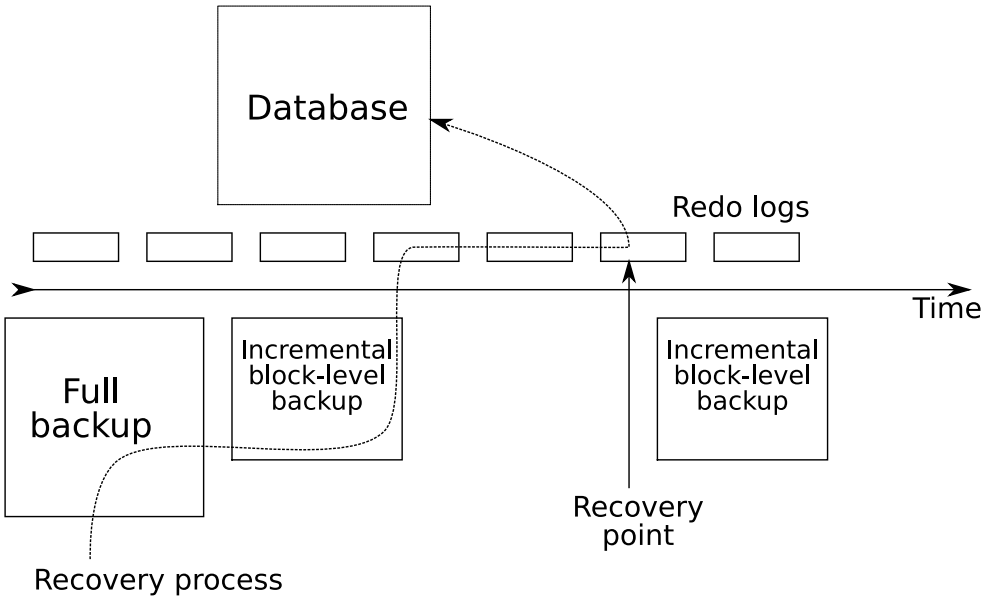


Figure 2.4: Database recovery using full, block-level incremental backups and redo logs

especially useful if the different parts are located in geographically distant places. These solutions are mostly used in heavily loaded environments where no other solution could provide satisfactory performance.

2.1.3 Snapshot Backups

Although the big databases (Informix, Oracle) provide good tools to make online backups without stopping the system for unreasonable amount of time, there are still many other products that do not provide such functionality, yet many company business plans require these applications to be available non-stop. To eliminate the need for long downtime, the administrators can stop the application, create a snapshot and start the application again. Then it is possible to backup the application data from the snapshot.

The application downtime is minimized, yet no modification of the business software is required. Software snapshots have some impact on the performance, hardware provided snapshots or split-mirror solutions can be used instead if the performance impact was unacceptable. A detailed description of snapshot functionality is provided later in the text.

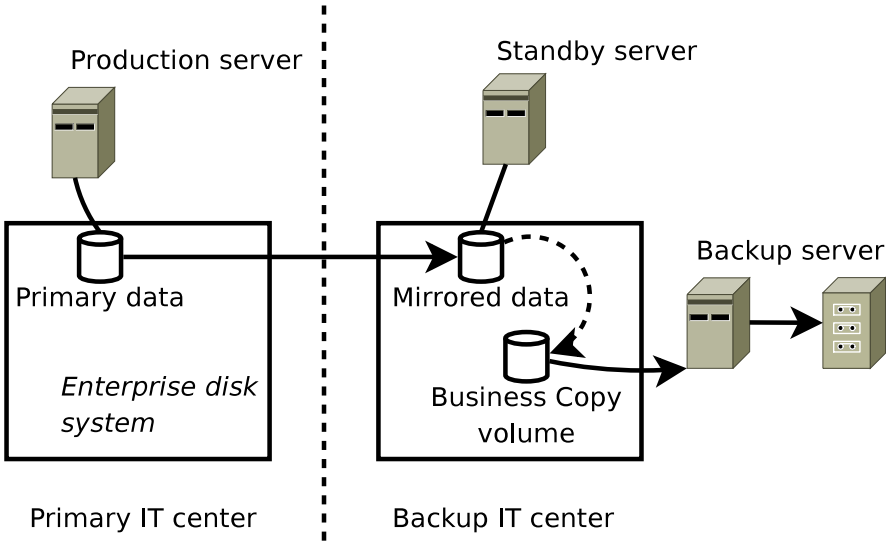


Figure 2.5: Geographical cluster with split-mirror backup using BCV

2.1.4 Enhanced Backup Solutions

Requirements for higher and higher amounts of data and lower backup window and impact on standard operation, lead backup software providers to seek other improvements. Many methods have been tried but only some of them have survived. The main criteria for survival seem to be ease of use and independence on operating system and thus easier maintenance.

There are several ways to increase the speed of backup and recovery – to move data faster, to move less data, let somebody else move data in order to reduce CPU usage. In order to move data faster, the backup devices are today usually connected directly to application servers. The Storage Area Networks even allow to connect one device to more servers provided that they do not access it simultaneously. The backup software commonly allows such configuration and allows tape drives to be shared between servers.

Recently, in order to speed up backups and reduce the necessary storage space, many vendors started offering so-called 'data deduplication' [6]. The storage software computes a hash of the block that is supposed to be saved and compares it to checksums of other blocks in its database. Should the hashes equal, the blocks are in many solutions assumed to be equal as well [7]. The assumption is that the probability of hash collision is significantly lower then a disk or tape error.

In order to move less data, block-level incremental backups have been proposed. Software agent monitors a disk device and tracks which blocks were changed. When a backup is made, only changed blocks are sent to the server or tape device. As many users require access to a particular file when restoring data, the backup software was enhanced to understand the filesystem structure and serve the clients data as if it was

backed up using regular file level backup.

To reduce the impact of backup procedures on all servers, these features were combined along with some HW support to facilitate LAN-free server-less backup. In the end, the server sends a list of blocks to the disk device; the disk device then sends the data directly to the tape, utilizing SCSI 3rd party copy command.

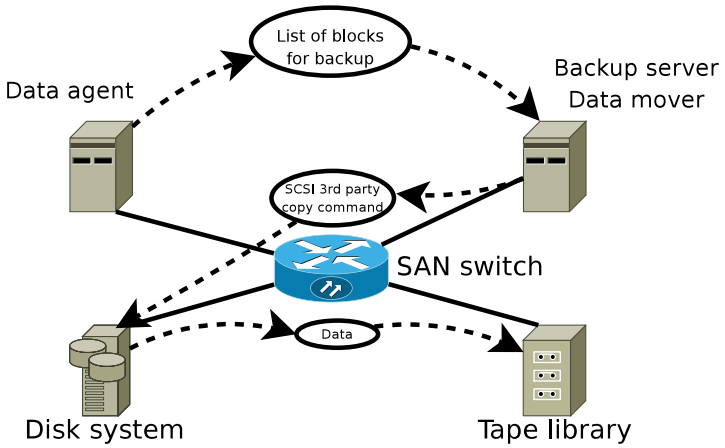


Figure 2.6: Server-less backup scheme

Although there are functional implementations, these enhancements are not in wide use today mainly because they proved to be too complex, require too many cooperating parties (OS vendor, HW vendor) and are hard to maintain.

2.2 Asynchronous Replication and Journalling

The term 'replication' usually denotes mirroring of data over LAN. To cope with longer latency on the LAN network, asynchronous replication was devised. All read operations directed to replicated device are sent to the faster locally attached device. Write operations are sent to all replicated devices. The write operation is acknowledged to the client operation when it reaches both local and remote disk with synchronous replication, as soon as it reaches local disk in the asynchronous mode.

The network stream of data sent by an asynchronous replicator is in fact a journal of write operations. It can be stored by the other side and when accompanied by information about time and possibly some marks denoting e.g. consistent application state, complete information exists to provide time accessible storage with very high granularity.

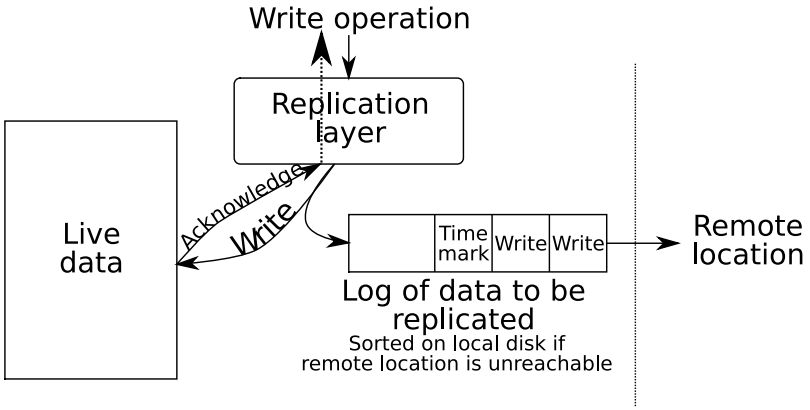


Figure 2.7: Asynchronous replication

2.3 Snapshots

Snapshots are probably the most widely used technique used to minimize backup window. There are many flavors of snapshots, most of them use 'copy-on-write' algorithm. A snapshot intercepts all write operations and before letting the write operation proceed, it copies the underlying block to some other storage. The user is then presented a a virtual 'snapshot' of the old data by mapping the read and write request to the correct blocks.

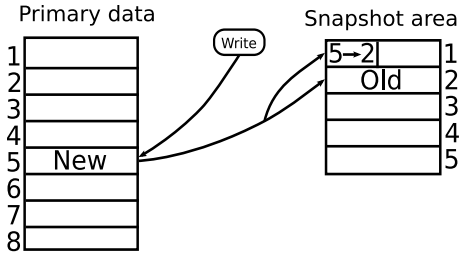


Figure 2.8: Copy-on-write snapshots

2.3.1 Disk-level Snapshots

The simplest form of a snapshot is a disk-level snapshot. It is supported natively by most operating systems and enterprise storage systems. Operating systems typically allocates a piece of disk that is then used as a copy-on-write area for both the writes to the device containing the live data and the writes directed into the snapshot area. The enterprise storage systems typically allocate a space that is equivalent to the disk with live data and only use a simple map that indicates which block has been changed.

Modern storage virtualization systems provide sophisticated systems of chained read-only *snapshots* and read-write *views* of the snapshot of the data. The idea is that while snapshots are used as a means of making immutable backups, the writable view can be used e.g. for testing.

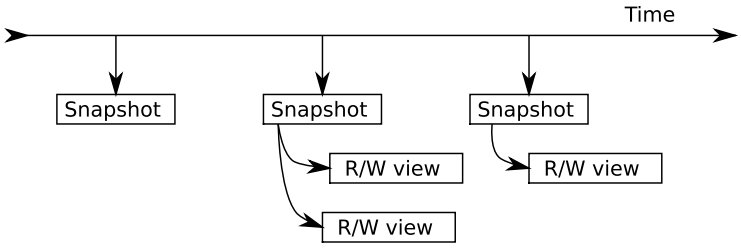


Figure 2.9: Snapshots and views

A big performance hit is hidden in the copy-on-write algorithm: in order to fulfill a write into the live area, the original block must be first read, copied to the copy-on-write area, then the new data can be written and acknowledged to the user. This greatly increases the latency and slows down the operations.

In order to provide faster snapshots, while maintaining reasonably secure data immune to loss because of power failure, it is possible to add disk journalling in front of the snapshot. The write operation is acknowledged to the writer as soon as the new blocks reach the journal. The copy-on-write operation is then performed in the background by the storage virtualization software.

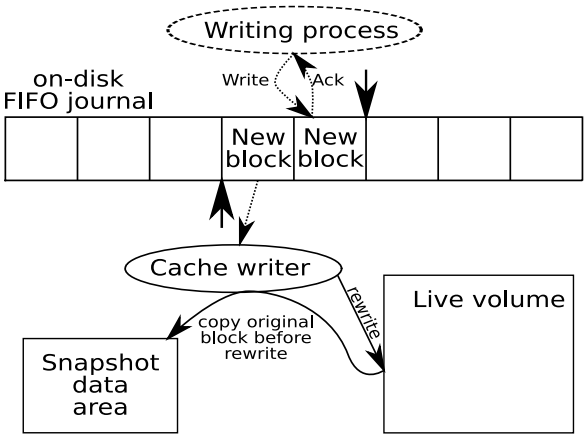


Figure 2.10: Journal to speed up snapshots

2.3.2 Continuous Data Protection

By combining journalling and snapshots we end up with so called Continuous Data Protection. Continuous Data Protection software allows users to access historical versions of data with almost infinite granularity. There are several products on the market, usually one-purpose appliances or parts of storage virtualization software that provide these services over the SAN.

Although there are probably several ways to implement continuous data protection, the combination of snapshots and write journal leads to a very simple and fast solution. Other solutions generally impose bigger penalty in terms of performance. These solutions are often connected to the servers as a second mirror that is only being written to; CDP functionality can be added to the production environment with minimal disruption.

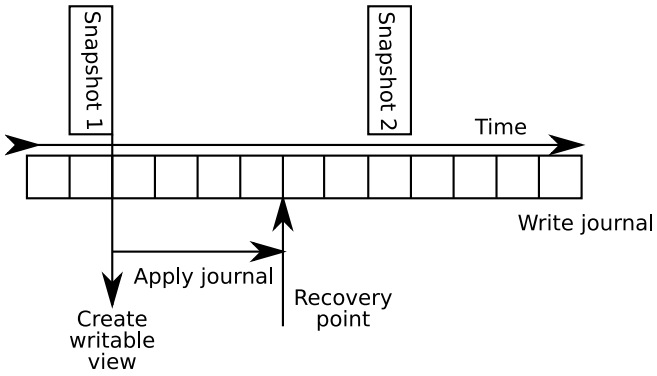


Figure 2.11: Continuous Data Protection

The Continuous Data Protection can be implemented in several ways. The most straightforward is a modification of some asynchronous replication software. Alternatively, it is possible to create a special purpose ‘split-write‘ disk driver that forwards the write operations to some CDP appliance. Probably the best solution is to use a storage virtualization layer as it requires no changes in the server operating system and thus minimizes administration requirements.

2.3.3 Filesystem-level Snapshots

There are several filesystems with versioning or snapshot support but not many are in wide use. Most of the filesystem (VxFS, ext3cow, WAFL, etc.) allow users to access only a read-only snapshot, in most cases to provide means to backup consistently live data. Ext3cow filesystem was specifically designed to store multiple versions of files because of backup and archiving purposes. Only Sun ZFS filesystem allows administrator to create a writable clone of the file system that directly shares data with the

original live filesystem and allows administrator to test things without changing the original data.

The layout of traditional filesystems is actually quite friendly to adding snapshot functionality. Creating a snapshot e.g. in ext3cow filesystem is by itself a very cheap and fast operation. Every inode is extended with information about snapshot epoch number and a bitmap indicating weather the data block has already been modified by this snapshot or if the copy-on-write operation should be started upon write request.

When a new snapshot is created, the system simply increases the snapshot epoch number. When a file is accessed, the system checks whether the file epoch number is smaller than snapshot epoch number — if it is, the inode is duplicated, it is assigned the snapshot epoch number and if a block is to be rewritten, a copy-on-write operation is performed.

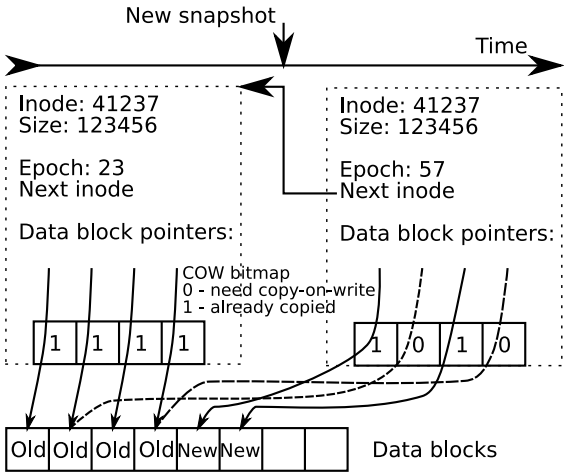


Figure 2.12: Inode cloning in Ext3COW filesystem

2.3.4 Open File Manager

Sooner or later the administrators come across an application that cannot be easily backed up using standard methods because the data is changed during a backup and the restored data is inconsistent and unusable.

For Windows platform the Open File Manager can be used to present different versions of files to different programs. The Open File Manager detects that a backup process should be given a snapshot of the data instead of live data and allows the backup software to create consistent backups even for applications that do not directly support it. Unfortunately, there does not seem to be any similar solution for UNIX machines, disk-level snapshots are often used instead.

2.4 Hierarchical Storage Management

A very interesting approach to storing data is a Hierarchical Storage Management. The idea originally evolved to provide storage system that could automatically utilize expensive and fast media for often accessed and important data and cheap and slow media for less frequently accessed data, while automatically providing migration of data between different storage layers. However, the most important feature of the HSM systems is that they practically eliminate the need to do backups.

The typical HSM filesystem stores the directory structure on the disk; for every file there is a structure describing the media and location of the content of the file. There can be more copies of the data for every file and when the user wants to access the data, the system can read the data from several different media until the request is completed.

In fact every file in the system is automatically backed up, but if a defined number of copies is made, there is no need to create more copies, as it is done with traditional backup software. In case of disaster, only the meta-data needs to be restored; the contents of the files can be moved to the fast disks later. A recovery from totally destroyed disk cache of a 50TB system is a matter of minutes.

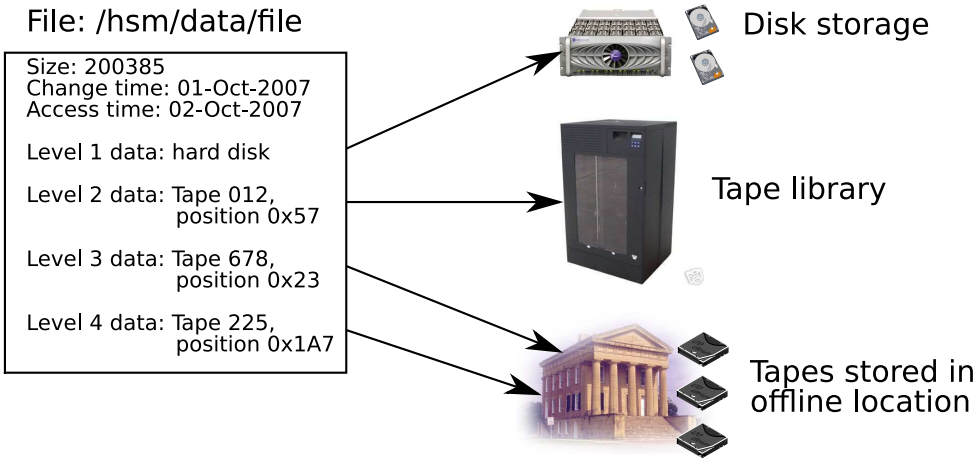


Figure 2.13: Hierarchical Storage Management

The HSM systems are not directly meant to provide access to older versions of the data but in fact many do as a side effect. The files are typically copied to a different storage within minutes since they were last modified. Data written to the tape is never rewritten. If users modify a file, new copies of the file are written to the tape. Accessing the old version of the file is thus just a matter of restoring the old meta-data pointing to the locations where older versions of the files are stored.

Some filesystems like XFS, JFS, IBM GPFS and VxFS support DMAPI – Data Management Application Interface which can be used to extend these filesystems with HSM capabilities.

2.5 Network Attached Storage

Network attached storage is a computer whose sole purpose is to act as a file server. It usually supports both Unix and Windows clients and functions to achieve high reliability of serving the data. In terms of time addressable storage, e.g. the NetApp NAS allows users to mount up to 256 snapshots over the NFS. NetApp has its own WAFL filesystem (write anywhere file layout), which supports read-only snapshots.

In order to create backups of NetAPP, the NDMP protocol was developed that facilitates server-less backup. The NAS server communicates with the backup server and sends the data directly to tape drives without passing the data through backup server first. Most current backup software supports this protocol.

2.6 Storage Area Network

"Everything is a network device" was probably a motto that has led to developing storage area networks. Or it might have been problems with SCSI bus and trying to connect one disk to two clustered servers. In any case, the storage area networks solve many problems and although at the beginning there used to be big problems with compatibility, today the SAN is a normal choice in enterprise environment.

The term SAN originally denoted mainly the technology built on Fibre Channel protocol. The protocol supports among others automatic address allocation and topology discovery. Every device has a unique 'WWN' – world wide name; moving device from one physical location to another does not usually require any reconfiguration of SAN routing.

Although some protocols that provide security and authentication in FC networks exist, they are not widely used. Thus, security in most FC networks is in fact non-existent and depending only on physical shielding of the SAN network from the outer world. The administrators use zoning both to eliminate unneeded disks to be seen on the servers and to stop servers from destroying data designated for other servers, but even the zoning is not — in certain instances — enough to stop a potential attacker.

The simplicity and proliferation of IP networks together with higher speeds and lower latency prompted creation of an iSCSI standard – SCSI over IP. The protocol is designed to facilitate connecting devices to servers using standard IP networks and provides some basic methods of authentication. However, the data is either sent cleartext or an IP-Sec protocol has to be used to encrypt the data, which adversely impacts the performance. Because of the security concerns, the use of iSCSI protocol probably remains confined to closed and trusted networks as a low cost alternative to FC

networks.

2.7 Storage Virtualization

With the introduction of storage area networks the difference between a storage system and a server became blurred. It became possible to write a simple program that would turn an ordinary server into an appliance that behaves like a SCSI device, it was possible to modify firmware in fibre channel switches to inspect and change the data flow between servers and hardware, the servers could communicate with other appliances not only over LAN, but they could simply use the fast SAN infrastructure as well. Everything just became network connected device.

The very first services that were provided by the storage virtualization appliances were very similar to the classical volume management. The virtualization appliances allowed administrators to utilize excessive capacity and easily add more storage in case of need.

The storage virtualization solutions can be roughly divided into in-band, out-of-band and solutions using intelligent switch firmware. With the in-band approach the appliance presents itself directly as a disk device, all data passes through the appliance. Out-of-band type of virtualization needs a special driver on the servers that communicates with the appliance and directs data on appropriate devices. The new SAN fibre channel switches allow to be reprogrammed so that they can do some tasks depending on the data that is received – they can function similarly as network address translation in LAN routers.

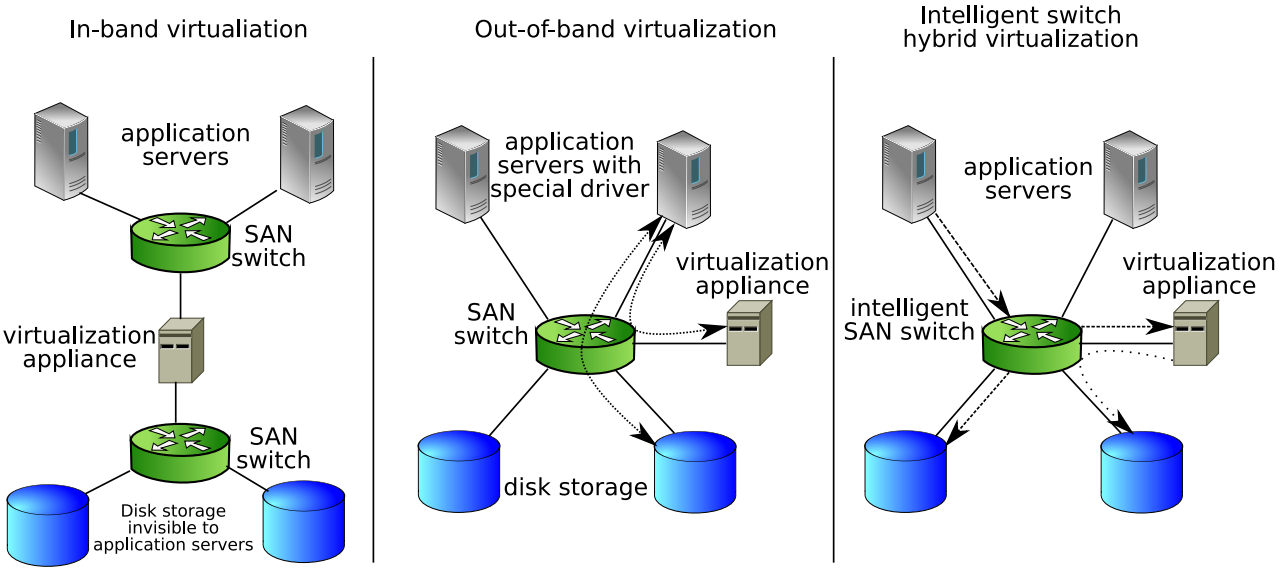


Figure 2.14: Different types of virtualization

The advantages of in-band solutions outweighed smaller latency of out-of-band solutions. Today, the storage virtualization solutions provide complex services including high availability, clustering, mirroring, several levels of RAID, synchronous and asynchronous replication, snapshots, views. In fact, the expensive one purpose enterprise storage systems are slowly being superseded by storage virtualization solutions.

Unlike out-of-band solutions, the in-band solutions somewhat improve the security in FC SAN networks. When proper zoning is employed, the servers can reach the disks only through the in-band storage virtualization appliance. This itself eliminates accidental overwriting of the disks pertaining to different server. It does not solve other security problems of FC technology, however implementing some FC authorization protocol that would prevent WWN spoofing and other attacks is much easier when interfacing to one appliance instead of multitude of different disks and storage systems.

The in-band solutions based on general purpose computers can easily provide not only SAN based storage services, but NAS services as well with the advantage of having all the high availability and the backup tools available. Such appliances really help reduce the task an administrator faces when ensuring business continuity in enterprise environment.

All that is needed to create a simple storage virtualization solution, is to take a Linux machine, installing Linux target framework (iSCSI target, FC target, SCSI target, etc.) and publishing some disks over the target framework to other machines. Linux already supports several levels of RAID, volume management, synchronous replication (drbd [4]) and is thus capable of providing storage virtualization solutions without much other work.

2.8 Classification of Time Addressable Storage

The Time Adressable Storage is typically classified according to two criterie:

RTO — Recovery Time Objective is the time needed to recover the system.

RPO — Recovery Point Objective is the amount of data that is allowed to be lost.

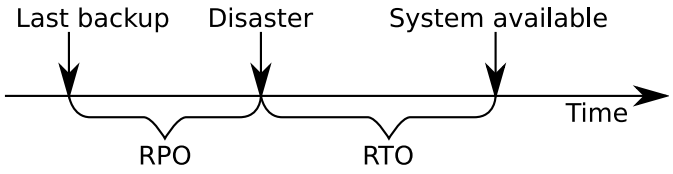


Figure 2.15: Recovery Time Objective, Recovery Point Objective

The optimal solution would not be expensive and would have near-zero both RPO and RTO. In the table 2.1 the most widely used techniques are summarized along with their RTO and RPO.

Method	**RTO**	**RPO**
Only full backups	long	long
Full and incremental backups	very long	average
DB full backups with archive logs	very long	very short
DB delayed replication	quick	very short
Block device snapshots	very quick	short
Block device delayed replication/journalling	very quick	extremely short

Table 2.1: Classification of Time Addressable Storage

Despite its very inconvenient classification, the classical backups are still widely used. The costs of implementation of the new solutions is still prohibitively high in many companies. However, as the price of many of the new solution declines, we can expect more companies to convert their Business Continuity strategies to use the new technology.

Considering the current development in the area of storage virtualization, it is very likely that the application-specific solutions, such as database delayed replication, will be abandoned in favour of storage snapshots and remote replication.

Chapter 3

Linux Storage Solutions

Today Linux OS is equipped with quite modern solutions to provide high availability of storage systems. Historically, the *md* driver provided capabilities of software RAID. Recently, in order to facilitate Logical Volume Manager (LVM), a *device mapper* layer has been added. The device mapper layer does not support raid4/5/6 functionality and mirroring had until very recent kernels been totally unusable and prone to lose data (unfortunately, this 'feature' was not loudly advertised).

In high availability environment the DRBD synchronous replication is being widely used. Although some functionality could be replicated with iSCSI and local mirroring, DRBD supports correctly cluster filesystems mounted on both sides of the replication. However, although this solution could be used to provide split-mirror backups, it is more suitable for high availability use.

Ignoring multitudes of ordinary backup software, there are a few commercial solutions available for Linux. Above all, any platform independent solution can be easily used, which includes basically all in-band virtualizators (FalconStor, Datacore SanSymphony, etc.). Some of these solutions are even based on Linux but they use their own proprietary tools to provide replication and snapshot services.

As for clearly software solutions, Linux LVM directly supports snapshots. R1Soft CDP [14] supports combination of snapshots and remote replication to build an interesting solution directly usable as a means of backup.

3.1 Logical Volume Manager

Linux LVM 2 is functionally identical to HP-UX volume manager. It is a set of user space utilities whose main aim is to produce virtual partitions spread over diverse physical disks. Such solution reduces administrative overhead and makes disk allocation and adding space to disks easier. The LVMs were actually a predecessor of the current hardware storage virtualization solutions.

The LVM stores its configuration information directly on the data disks. When the configuration is changed or the system is booted, the LVM utilities update the kernel

device mapper tables to reflect the new block mapping. LVM presents administrator interface comprising Physical devices, Volume groups and Logical volumes. Each volume group is comprised of one or several physical devices and provides one or more logical volumes (partitions).

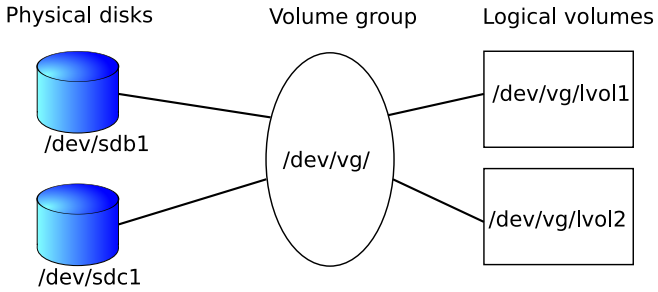


Figure 3.1: LVM architecture

Internally, the things get a little more complicated. Each logical volume is comprised of several segments. Each segment represents some part of the logical volume, possibly striped over several physical disks. When a logical volume is extended, a new segment is added.

The segments contain a 'type' property and are the main means of extending LVM with new functionality. When a segment with new type is loaded, a type-specific code is executed that can modify attributes of the rest of the logical volumes. When a new snapshot is created, a new logical volume with one segment of the 'snapshot' type is created. This segment contains special attributes — `origin` and `cow` — identifying original live volume and area for snapshot data respectively.

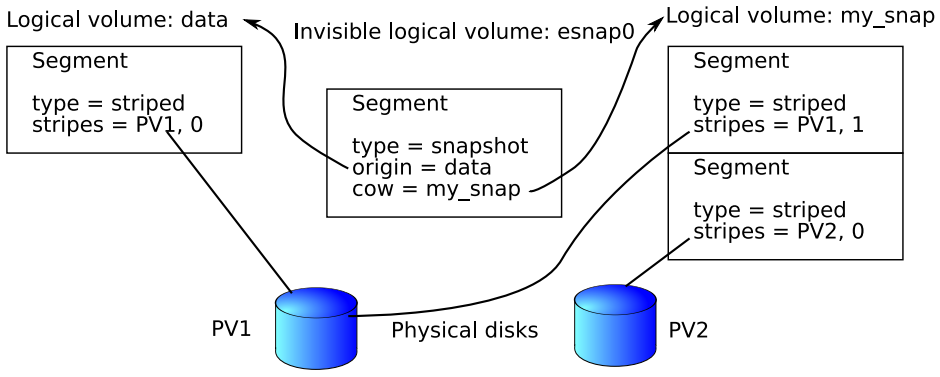


Figure 3.2: Snapshot representation in LVM

3.2 Device Mapper Architecture

To facilitate logical volume manager a device mapper API was added to kernel. The API consists of 12 functions whose main goal is providing an infrastructure to create virtual devices that are assembled from parts of other devices. The sole aim of the main function — `map` — is to modify the sector and device numbers of the read/write request and redirect the operation to another device. To support different types of virtual devices — encrypted device, mirrors, snapshots, alternate path switching on SAN — the `map` function can modify the data that are sent to the disk, decide not to send the data and do some pre/post processing.

Each device mapper device is described by a table defining which part of the device is handled by which *target*. When a user accesses certain part of the device, the device mapper routes the request to the *map* function of the appropriate target.

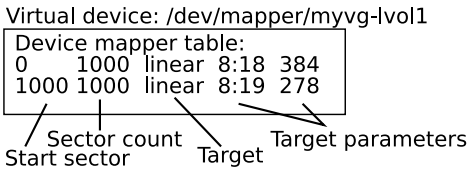


Figure 3.3: Device mapper table

Moder UNIX logical volume managers are able to change size of the data volumes without unmounting any filesystems. The table describing the virtual device can be changed, provided there are no pending I/O operations on the device. In order to enlarge the disk, add mirroring or do any other operations, the device is first suspended, then the tables are switched for new ones and the device is resumed. Suspending means waiting for all I/O operations to complete and queueing all new incoming operations in the device mapper layer. The device mapper architecture provides 2 functions – `presuspend` and `postsuspend` — that are called before and after the device is suspended respectively.

Sometimes it may be needed to suspend a device which cannot complete the I/O requests while not letting the I/O requests fail until the device is resumed again. This is in particular a problem with multipathing device. The multipath device creates one virtual device connected to the system over several different physical paths (usually using SAN network). When a path fails, it just switches to another one. However, if all paths fail, the I/O operations get queued in the multipath layer. The user can reconfigure their systems and add a new path. Unfortunately, in order to add a new path to the device, they would have to suspend the device, which is impossible without returning all pending I/O with an error.

To solve this problem a *noflush* suspend was implemented. When the device mapper target detects that it is in *noflush* suspend mode, it can 'push back' the pending I/O requests that had not been sent into lower layers. These requests end up in the device

mapper queue and are resubmitted when the device mapper target is resumed again.

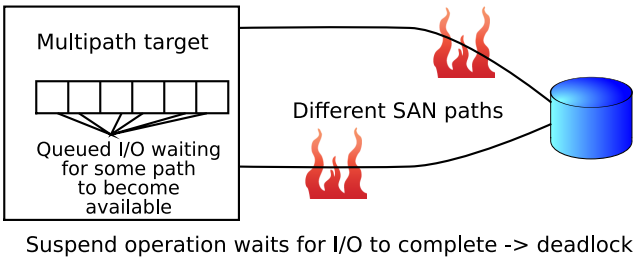


Figure 3.4: Multipath deadlock on suspend

The device mapper layer is built in such a way that an error in a table change should be reversible. When a table is changed, a new temporary table is constructed first. Appropriate constructors of the target devices are called. Only when the device is resumed, the old table is destroyed, all destructors are called and then the resume method of each newly configured target device is called.

Sending messages from user-space to device mapper target can be accomplished through the `message` function, user-space can obtain more information from device mapper target through `status` function. In order to send messages from device mapper target and user-space, the target code can set an 'event' on the device which is monitored by a *dmeventd* daemon.

LVM uses special naming convention when creating device mapper devices: `volume_group-logical_volume[-suffix]`. All device mapper devices are created in `/dev/mapper/` directory, the LVM creates symbolic links from the appropriate `/dev/vgname/` directories.

3.2.1 Device Monitoring

The device mapper devices can be automatically monitored by the *dmeventd* daemon. This daemon is part of user-space *libdevmapper* library and can easily be extended with dynamically loaded libraries provided by e.g. LVM. The daemon monitors a device mapper device and when an event is received (optionally when timeout expires), registered library function is called.

LVM currently provides monitors for a mirror target and a snapshot target. When a failure occurs in mirrored environment, the failed mirror is removed from the logical volume according to specified policy. When the snapshot is getting full, a message is written to syslog to alert the administrator that he should extend the snapshot volume.

3.2.2 LVM and Snapshots

As users usually want to create a snapshot of a device that was not previously prepared for it, LVM must completely change mapping of devices when the snapshot is being

created. The mapping that comprised the original device is recreated as a new device with a suffix `-real`. The mapping that represents the area allocated for snapshot data is created as a virtual device with a suffix `-cow` (as in copy-on-write).

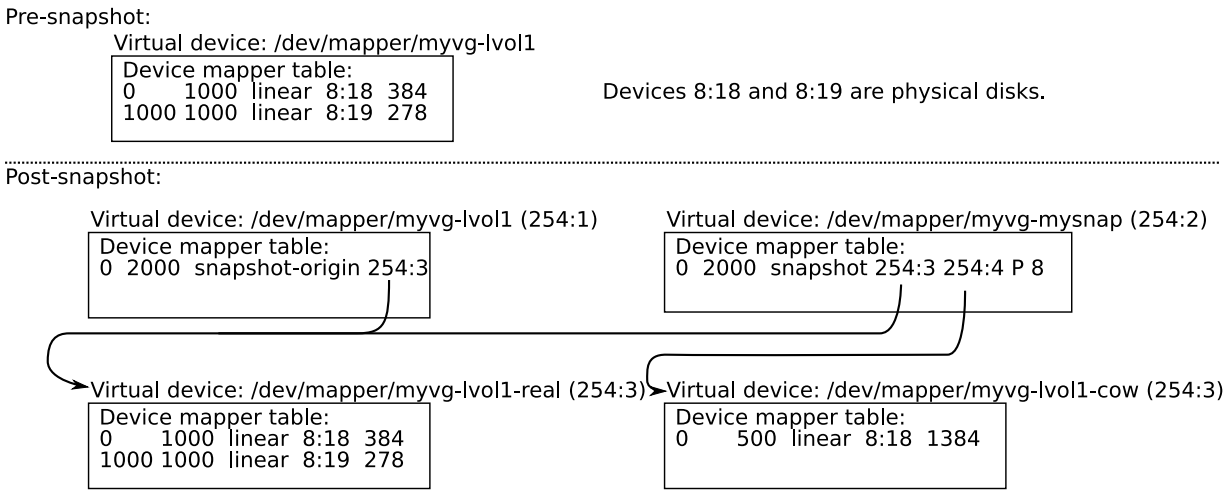


Figure 3.5: Device tables belonging to snapshot

3.3 Kernel Snapshot Driver

The snapshot implementation in Linux uses a very simple and straightforward design. Every snapshot is designated a special disk area (COW) to keep track of the original content of the disk before a change occurrs. The snapshot uses a copy-on-write algorithm. When the snapshot is created, every write to the primary disk is caught and a copy of the original data is appended to the COW area. If the data has already been changed since the snapshot was created and the original block already resides in the COW area, this step is skipped. Then the meta-data about the location of the data blocks is written into the COW area. Upon completion of this copy, the write operation of the new data can proceed.

Multiple write operations can be handled simultaneously and the meta-data being written to the disk can be bundled together for the simultaneous operations taking place. Because the callback function called upon completion of an I/O operation is called in an atomic context (no mutex can be locked and memory can be allocated only in GFP_ATOMIC mode), special kernel thread — `kcopyd` — was created that helps with the copy-on-write operation. The `kcopyd` is simply given a sector address of the source and destination disk and a callback function; when the operation is complete, the callback function is called in normal, non-atomic mode.

The COW area is being filled sequentially with the original data and data chunks containing information about which chunks have been changed and where the original

data reside in the COW area. One meta-data chunk contains information about the location of the subsequent data blocks. It is possible to access the snapshot disk in normal read-write mode, the writes directed to the snapshot area will be tracked in the COW area as well.

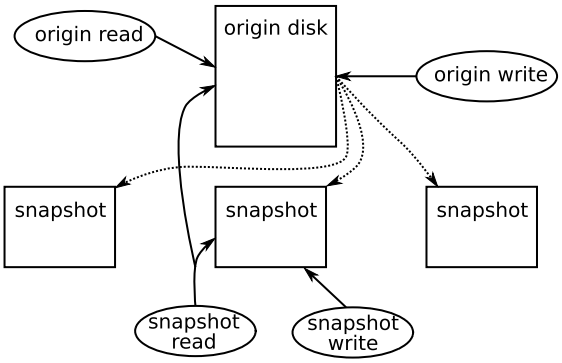


Figure 3.6: Independent writable snapshots

All snapshots are completely independent, which means, that the number of physical write operations is linearly dependent on the number of simultaneous snapshots. Performance is somewhat impacted, however all these operations can be made in parallel. When the COW area becomes full, the snapshot is disabled and becomes inaccessible. However, before the COW area becomes full, the user may enlarge the area. LVM currently supports this operation.

The portion of data that is copied when a copy-on-write operation occurs, a *chunk*, can be user specified. It must be a multiple of page size and larger then the sector size of the underlying device. In older kernels larger sizes were recommended because of high latency, but with the introduction of `kcopyd` thread the smallest size is recommended.

For every snapshot the driver holds in memory hash table of changed blocks and their redirections into the COW device. The disk format of the COW device is fairly simple: the COW device is divided into user defined snapshot-specific chunks. First chunk contains a simple header that is followed by data that was changed on the original device interleaved with meta-data blocks containing information about the changed blocks and where it was copied. The snapshot driver developed in this these uses practically the same format (see fig. 4.2).

Since the introduction of the `kcopyd` kernel thread the snapshot driver has not needed to serialize the writes — copy data + write meta-data, copy next data + write meta-data — but it can copy the data in parallel and write the meta-data once the meta-data block is complete or there is no more data to copy. This behavior slightly increases the write latency, but evading serialization drastically increases the throughput.

There is still some serialization built into `kcopyd`. When a meta-data segment gets full, the meta-data must be written to disk. Until the meta-data is written new copies

of data will not be started. For 4K chunks size the segment contains 512 entries. When a meta-data block is written, next block is automatically filled with zeros.

When a new `snapshot` target is constructed, the COW area is checked. If the first block is filled with zeros, a new snapshot area is initialized. If a magic number is found, the content of the COW area — the chunk redirections — is read from the disk. The process just reads meta-data blocks from the disk until a partial meta-data block is found. There is some potential for incorrect operation of this routine when the computer crashes after writing meta-data block and before writing zeros into the next meta-data block.

The implementation supports a *transient* snapshot as well. The *transient* snapshot is not designed to survive reboot and as such the meta-data operations can be omitted and the write latency decreased.

The Linux implementation in the 2.6 kernels is located in the *drivers/md* directory and consists of files *dm-exception-store.c*, *dm-snap.c* and *dm-kcopyd.c*.

3.3.1 Disadvantages of Current Snapshot Implementation

The snapshots in current implementation are totally independent. This approach suffers from several disadvantages compared to other snapshot implementations available.

- With the increasing number of snapshots each copy-on-write operation must be recorded into more COW areas, which somewhat reduces performance.
- As each copy-on-write operation is written into multiple snapshot areas, a lot of redundant disk space is wasted.
- Writes to the snapshot are permanent, i.e. the original snapshot is not retained when the snapshot is used e.g. for testing. This makes it rather unsuitable for backup purposes.
- There is no automatic extension of snapshot area should it become full. Although some scripts may be defined to watch syslog and extend the snapshot as soon as the utilization of snapshot area is getting high, it is neither systematic nor reliable.

The standard Linux snapshot driver is suitable as a means of snapshotting the data before making a backup, especially when using the ‘transient‘ option. It is not suitable as a means of making large number of backups periodically as having these snapshots available adversely impacts performance and is needlessly space inefficient.

Chapter 4

Design of a New Snapshot Target

As current Linux snapshot driver is unsuitable to facilitate time accessible storage services, a new snapshot driver is proposed. The following goals should be met:

- Provide standard snapshot-view model, so that it could be easily used for backups and testing
- The impact on performance must be independent of the number of snapshots
- Integrated into LVM, so that the users need not learn and install other complicated software
- The snapshots should automatically extend themselves, so that the snapshot space is well utilized and the administration is simplified

The implementation of a new snapshot thus requires:

- Implementation of a kernel driver
- Extending *libdevmapper* library to accommodate new device mapper targets
- Extending LVM to support new commands to create, remove and extend snapshots and views
- Create an extension library for dmeventd that would automatically extend the appropriate snapshot/view disk area when it becomes full

4.1 Kernel Driver Design

Time accessible storage based on disk devices takes generally 2 forms - continuous data protection (see chapter 2.3.2) or snapshot/view design. Although CDP can possibly be solved by journal indexing, a very reasonable implementation of CDP is nevertheless based on snapshot/view model. Many existing so-called near-CDP solutions employ a model that is based on a large number of snapshots, so that a very high time granularity of any-point-in-time restore is achieved.

Considering that the snapshot/view model satisfies most needs for time accessible storage, it can be easily extended to provide data for delayed asynchronous journalling and can serve as a basis for CDP, it will be the basis for this kernel driver.

There are 2 possible ways to provide snapshot services: copy-on-write and redirect-on-write. Copy on write operation copies original data before it is changed on the *origin* disk, while redirect-on-write approach redirects the writes and subsequent reads to a new unallocated space.

The portion of data that is being copied may be larger then the block size of a device and will be called a 'chunk'. The disk with production data will be called an *origin* disk and the snapshot area containing copies of the original data will be called a *COW* (copy-on-write) area. This terminology is consistent with the terminology used in the present Linux driver.

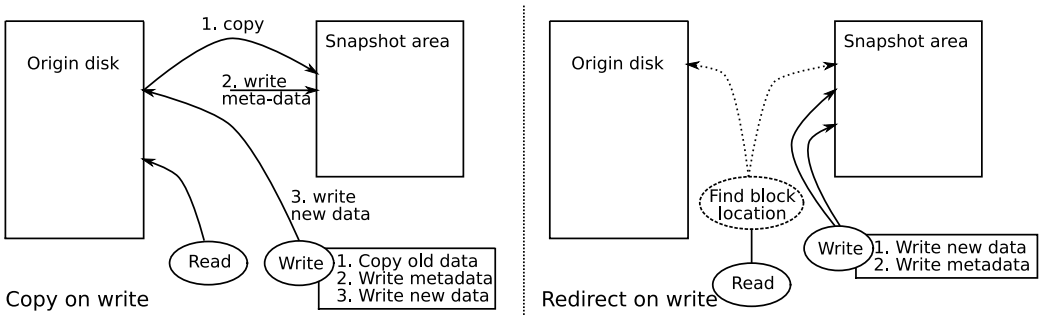


Figure 4.1: Copy on write vs. Redirect on write

Both solutions differ in implementation of read, write operations, remove snapshot operation and in different requirements on memory.

Read	Directly from origin disk
Write	Search in-memory map, if not found copy block (1 copy (read then write) then meta-data write then 1 write of new data)
Remove snapshot	Free memory structures
Memory footprint	Table of changed blocks of last snapshot, tables of relevant snapshots when accessing view
Origin disk state	Newest data
Smaller writes then chunk size	Simple

Table 4.1: Copy on write

It is evident that the copy on write approach has a very expensive write operation, which is postponed to the removal of the snapshot in the redirect on write method. On the other hand, the copy on write method requires smaller memory footprint, has very simple read operation and keeps the origin disk in a consistent and most recent state.

Read	Search in-memory map, read from appropriate place
Write	Search in-memory map, if not on newest snapshot, write (1 write of new data then 1 meta-data write)
Remove snapshot	Rewrite snapshot data into original disk, do consolidation.
Memory footprint	Tables of all changed blocks of all snapshots required for read operations
Origin disk state	Data at the time of the oldest snapshot
Smaller writes then chunk size	Usually not supported or supported through a copy-on-write method

Table 4.2: Redirect on write

In terms of performance the redirect-on-write seems to be significantly better, but the amortized time of such solutions must be taken into account; the *snapshot delete* operation is usually so expensive that the performance is on par with the copy-on-write systems enhanced with write journal. The redirect-on-write systems are usually based on a filesystem concept. For example the NetAPP NAS appliance can serve as a storage virtualization appliance by exporting files on its WAFL filesystem as disk devices through Fibre Channel or iSCSI protocols.

The copy-on-write performance problems are not necessarily tied to throughput, but mostly to drastic latency increase. The 4 operations during write operation must be serialized and while we can expect the write operation to be performed extremely fast due to storage system write cache, the read operation must be typically directed to the physical disk.

The redirect-on-write systems do not handle well the situations when a write operation smaller then a chunk is executed. The system would either have to support different sizes of blocks, it would have to use a very small chunk size or it would have to disable the smaller blocks altogether. It is currently impossible to affect a *sector size* (the smallest block that a device allows) in the current Linux device mapper layer.

Considering the above mentioned constraints, a copy-on-write snapshot seems to be a good choice. It is simpler to implement, less error-prone, the data is not scattered over large disk area. Although the write latency of such implementation is higher, there are some solutions to alleviate this problem.

4.1.1 Copy-on-write Snapshots

The copy on write operation starts a copy of the original data to the snapshot area. When the copy is complete, meta-data information to the snapshot area is written. In this stage multiple copy operation may be grouped together so that all copy operation need not be serialized and one meta-data write is made when the group of copies is complete.

There are several options on grouping the meta-data writes. The origin Linux snap-

shot driver waits until the meta-data block is full or there are no pending operations. The other possibility is to start writing the meta-data block as soon as there is something to write, so that the write operations would be acknowledged as fast as possible. The first way maximizes throughput, the second latency. Although in some cases the first way may be preferred, in todays world of storage systems with battery backed write cache, the second solution would probably yield better results.

The current Linux snapshot driver has some serialization built in the meta-data area. Next snapshot area cannot be written until the last one is written on the disk and next block filled with zeros. First, this is a potential problem in case of a power failure, second this adds to the latency already imposed by the copy-on-write operation.

Ideally, an unlimited number of meta-data segments could exist in memory and be written to the disk simultaneously. However, considering that this is a disk target which should allocate the least memory possible, we are not left with many options. Only one segment can be being written to the disk, however the potential problem with a power failure was fixed and some more asynchronicity was added.

The on-disk format practically the same as with the old snapshot driver. A partially written meta-data segment indicates the last written meta-data block.

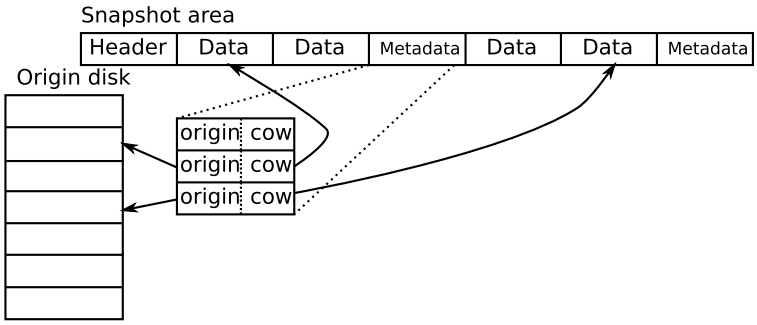


Figure 4.2: On-disk data structure

The redirection information is added to the meta-data blocks in the order of acknowledgment of writing the data blocks, not necessarily in correspondence with the position in the snapshot area, as shown in figure 4.2. It is much simpler to implement, slightly lowers the latency, but it has higher space requirements; the meta-data area can contain less records. The meta-data areas are scattered on the disk evenly, each is preceded exactly by the same number of records as it contains.

There are several in-memory structures required to keep the snapshot operation going. First, there must be some structure keeping track of the copied blocks. When a user wants to access the snapshot data, they consult this structure and if a match is found, the data is read from the COW area. If a user wants to write to the *origin* device, the table is consulted as well. If a match is found, a new copy is not created and the write is directed into the *origin* disk. In the Linux snapshot driver the redirection information is called an `exception`.

The `exception` structures in the original snapshot driver are kept in a hash table with maximum hash size of 12 bits. Considering the size of current disks compared to typical chunk size of the magnitude of a page size, such hash table is only marginally better then a linear list. With every write operation it is necessary to search for a corresponding `exception`. When one is not found then one will be added to the tree. The logarithmic complexity of the red-black tree will be significantly faster for large number of exceptions.

The second structure holds copies in progress — `pending_exception`. When a copy-on-write operation is started, the `pending_exception` structure is added to a table of pending exceptions and the write operation is added to a list of pending I/Os waiting for the copy operation to finish. When another write operation is directed into the same chunk, it is simply added to the list. The appropriate structure to hold `pending_exception` structures is a hash table. The table will not hold too many entries and the information is stored only temporarily.

When a copy operation is completed and the meta-data is written on the disk, the delayed I/O operations linked to the `pending_exception` structures are started.

4.1.2 Snapshot Chaining

The original Linux implementation treats all snapshots independently. If we treat the snapshots as incremental copies, we can reduce the performance problems associated with updated all snapshots significantly; only the last snapshot needs to be updated.

The memory footprint can be reduced too, as only the last snapshot data needs to be present in memory. All other snapshot tables need to be loaded in memory only when access to the snapshot is required.

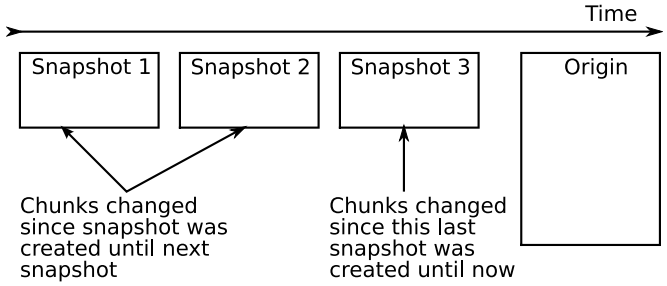


Figure 4.3: Snapshot chaining

Snapshot Read Operation

When the data from a snapshot is being read, the newer snapshots must be traversed to check if an old copy of the data is not found. If it is, the read operation is simply

redirected to the correct snapshot area. However, if no snapshot exception is found, the read operation must be redirected to the *origin* disk.

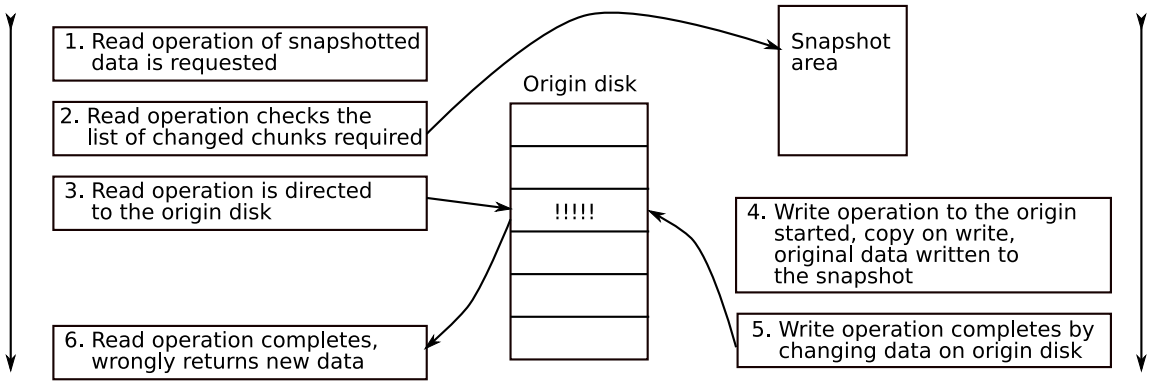


Figure 4.4: Race condition when reading from snapshot

When accessing the *origin* disk a race condition may occur; the read process may be started when the origin disk contained old data, but may be completed with newly rewritten data. There are 2 evident ways to evade the race condition — one of them is to check if the correct data was returned after the read, the other is to postpone the write operation until all reads are complete. This race condition was present even in recent Linux kernels, it was finally solved using the second approach. Performance-wise this is a better solution as the exception table does not have to be searched several times during every read operation. In terms of memory allocations, the implementation must allocate memory in both cases.

The concrete implementation is actually very simple — the `pending_exception` structure is used to track not only the writes to the origin device, but also the reads from the snapshots. A counter of in-progress read operations is kept. The write operation simply starts the copy-on-write procedure. When the copy is completed it checks the `pending_exception` for read operations in progress. Should some exist, the following writes are postponed until all read operations exit. When the copy operation is complete, new operations are directed to snapshot area, as shown in figure 4.5.

4.1.3 Adding and Removing Snapshots

When the user is creating a new snapshot, he often requires it to be created in an exact moment relating to the application data. However, a lot of data can be stored in application caches and a lot of data can be in the middle of being written to the disk.

In order to create a snapshot most systems first suspend the disk — wait for all the I/O operations to complete. This operation ensures that the data on the snapshot are at least as old as when the snapshot was made, not older. New snapshot can be specified by suspending the device mapper device, loading new device table updated with the new snapshot information and resuming the device again.

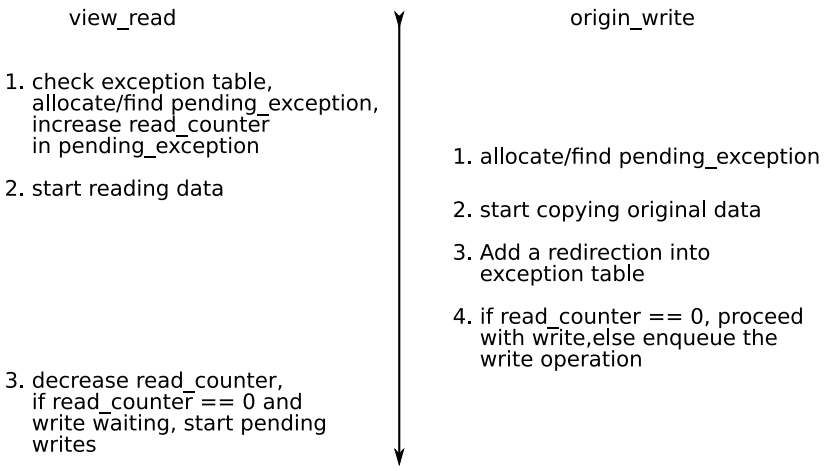


Figure 4.5: Obtaining correct data when reading from view

Some filesystems support a so-called *lock* operation to ensure that all data is flushed from the filesystem buffers to the filesystems and to ensure integrity of non-journalled filesystems. The *lock* operation should generally be called when creating a snapshot. It should not be called when removing a snapshot as it would adversely impact performance. Locking and unlocking filesystems is specified when suspending a device, the LVM layer should set the suspend flags accordingly.

Removing the snapshot can be done similarly as adding — suspending device, changing the table and resuming it again. The device snapshot area is simply removed from the internal structures. However, because there may be some view using the removed snapshot, a reference counting must be implemented. The LVM utilities should ensure that the kernel is correctly configured, nevertheless the kernel should be immune against such behavior.

4.1.4 Views

View is a writable snapshot. The implementation is very simple — another snapshot is inserted in front of the readable snapshot and there is a slight change in the semantics of the write and read operations.

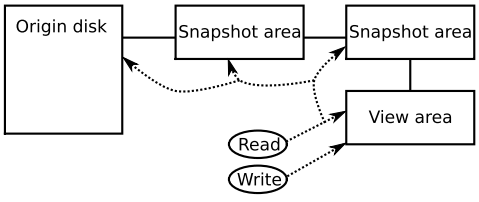


Figure 4.6: View read and write operations

The read operation first tries to find the data in the view snapshot area then it continues as a standard snapshot read operation, care must be taken to avoid race condition with origin writes.

When the write operation is required, the view snapshot area is searched for existing `exception`. If one is found, the chunk was already written and the write can be directed into the write area. If it is not found, a copy must be made. The source of the copy operation can be any snapshot device or the origin device, effectively executing a read operation. Again care must be taken to read the correct version of data from the *origin* device.

4.1.5 When the Snapshot Becomes Full

The COW area is typically fairly smaller then the *origin* device. There are several possible courses when the COW area is filled. The original snapshot device marks such snapshot as invalid. A daemon may be created to monitor the snapshot device and extend it before it gets full, however such solution may easily fail under higher load. This is not a viable option for time accessible storage.

A better solution would be to queue the write operations, signal to userspace volume manager that the snapshot is full and wait until the snapshot is expanded. This is fairly easy for the snapshot areas tied to the origin disk — the underlying device holding snapshot data can be simply enlarged and the kernel driver notified to process the I/O operations queueing for space to become available.

Given the LVM constraints, similar approach is somewhat harder to implement for the view. When extending a view device, the LVM suspends the view device. However, the suspend operation would deadlock (similarly as in multipath target, see chapter 3.2). Therefore the *noflush* suspend is used. When a *noflush* suspend is detected, all queued I/O waiting for a space to become available is 'pushed back' from the view target back to device mapper and the device is suspended.

This feature is further used when user requires removing a snapshot from the origin device while the origin device has some enqueued I/O operations waiting for available space. The enqueued I/O is simply pushed back, the device is suspended and the snapshot is removed. This works even when removing last snapshot which removes the snapshot target completely from the data path.

4.1.6 Journal, Data Consistency

Before considering different possibilities of modifying the copy on write or redirect-on-write approaches, an important constraint regarding data consistency should be mentioned. The disk storage should in general be safe against a power failure. The file systems and databases are written based on the premise that all acknowledged disk operations were written to the disk while some non-acknowledged operations may have been written to the disk.

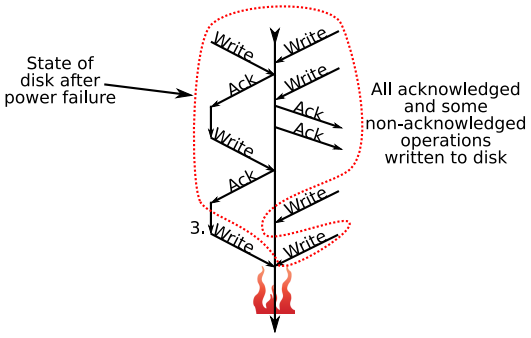


Figure 4.7: Consistent states after failure

When considering speeding things up, we have to consider if we want to keep the *origin* device in consistent state at all times. A journal can be used to first cache and acknowledge the write operations, while the copy-on-write operation is postponed until later. However, the journal may cause the *origin* device to be in an inconsistent state.

Today both software and hardware disk subsystems support parallel operations. For example the SCSI layer supports so-called tagged queue commands that allow the host computer to issue multiple read and write operations, the disk storage decides the optimal order and then notifies the OS when the read or write operations are completed. The same can be performed on the software level with different algorithms optimizing the disk head movement.

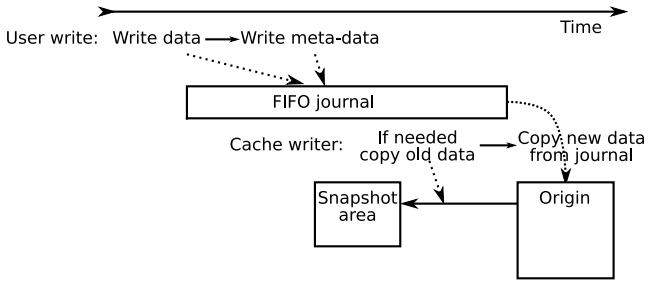


Figure 4.8: Steps when using journal

As described on figure 4.8, there are many operations that can be performed in parallel that increase throughput.

- The actual copy operation can be started immediately when the write operation is received.
- When a copy-on-write operation is not needed because the block was already modified since last snapshot, the journal write can be skipped.
- The journal writer process can perform the copies in parallel.

Starting the copy operation when the write operation is received is very simple and straightforward and causes no side effects. When the copy operation is complete, the journal writer process is notified that it can continue flushing the journal into the origin disk.

Skipping the journal write will improve performance, however it would directly cause the data on the origin disk to not comply with the constraint of time-consistent data. Additionally, the journal would not contain all operations and would not be usable as a source for a possible remote replication or any-point-in-time recovery. Although this functionality was not intended to be a part of this thesis, it was decided that this optimization will not be used.

The data written to the journal must be later rewritten to the *origin* device. The original Linux snapshot driver contains a `kcopyd` process that is submitted the blocks to be copied and starts all operations in parallel. If 2 writes of the same block of data appear in the journal, we may end up with the wrong copy on the disk and data corruption. Therefore, copying data from the journal to the *origin* disk must be serialized whenever duplicate blocks are found.

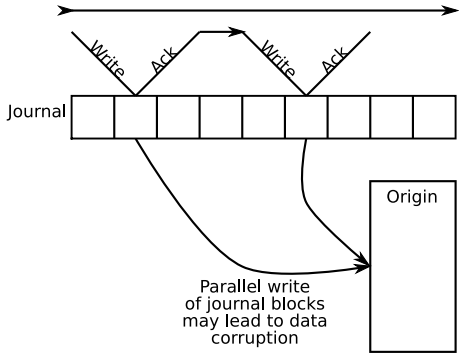


Figure 4.9: Data corruption/inconsistency with parallel journal write

If the software writing to the journalled disk depends on serialization of some writes, the parallel journal write may not preserve the serialization on the origin disk. If we wanted to enforce consistency of the data on the *origin* device, we would have to introduce some serialization in the journal writer, but retain some parallelism because of performance constraints. Serialization barriers would have to be included in the journal stream. Each write operation is acknowledged after data write and meta-data write, however multiple write operation may share one meta-data write. Therefore, this acknowledgment would be a place to add a serialization barrier.

We need to decide whether the *origin* disk should be consistent at all times or if the journal is an indivisible part of the data and the *origin* disk would be unusable without the journal. From the hardware point of view it is very unusual that a part of a disk becomes corrupt. An all-or-nothing scenario is much more frequent. Additionally, even if the journal is corrupted, the snapshots are still perfectly usable and

consistent. This snapshot uses the following strategy:

- All writes to the *origin* disk are directed to the journal first. This allows the journal to be used eventually for any-point-in-time recovery and other purposes.
- The consistency of the *origin* disk area is not preserved at all times in order to speed up the operations. The *origin* disk must be always accessed through the device mapper virtual device that ensures, that the user accesses consistent data.

The snapshot device must support a read operation. The read operation will be simply directed to the *origin* device if the block does not reside in the journal. If the block resides in journal, we may decide to read the block directly from the journal. Unfortunately this would be ridden with problems: firstly, the block written in journal may be smaller then the block size requested by the read operation; we would have to change the read operation to the partial reads scattered around the disks. Secondly, the journal is circular and we would have to keep track of the read operation so that we would not rewrite the data being written if the journal was getting filled. The safest way is to postpone the read operation until the chunk from the journal is flushed onto the origin disk.

4.2 Device Mapper Kernel Device — Implementation

The kernel module registers with the kernel three different target devices:

esnapshot-org — target that is used instead of the primary data device, it handles writes and copy-on-write operations appropriately

esnapshot — dumb target representing snapshots and journal

esnapshot-view — target representing writable view based on a snapshot

4.2.1 Target esnapshot-org

The target *esnapshot-org* is a core of the whole process. When a new *esnapshot-org* target is created, a `dm_origin` structure is allocated and registered as an *origin* for the particular disk device in a list of all origins.

For snapshots that belong to the *origin* device a `dm_cow` structure is created; it contains mainly redirection tables and queues for I/O operations when the assigned snapshot area becomes full.

The *snapshot-org* target must be constructed first, however it is not possible to assure in the LVM that the destruction occurs in correct order. Apart from that, when the target is being reloaded e.g. in order to create a new snapshot, the new target is created before the old one is destroyed. Simple reference counting system had to be employed in order to assure that the memory is not released before the last target using the memory structures is destroyed.

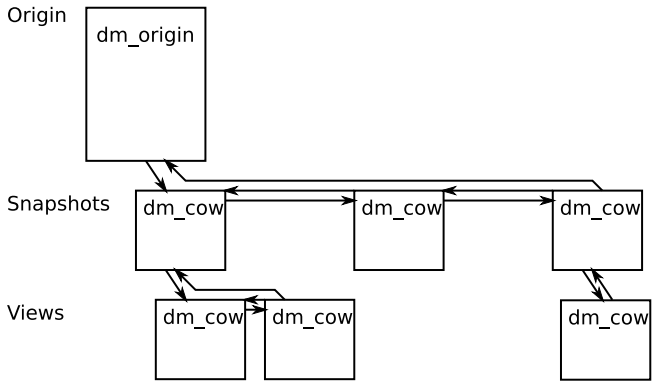


Figure 4.10: Structures representing the snapshots and views

Device (Re)loading

The snapshot construction code accepts following command line:

```
... origin_device chunk_size journal device
                    cow_dev [cow_dev2 [...]]
```

When a completely new *esnapshot-org* target is being created, the constructor simply opens all the COW areas and checks the headers. It starts with the oldest one and expects that the serial numbers are increasing. The most recent snapshot is checked for a magic number at the beginning. If a COW signature is not found, the COW device is initialized.

Creating the new snapshots or removing the old ones is performed by reloading the *esnapshot-org* with appropriate arguments. When the target constructor detects that it is being reloaded, it only checks that the specified devices match those already loaded in kernel and adds and initializes new snapshot area, if one is being added.

In order to reduce memory footprint, the exception tables in the `dm_cow` structures use lazy loading. Only the most recent snapshot has the tables loaded. Other tables are loaded on demand when the user requests access to a view that needs them.

Open Device Management

The device mapper layer contains special functions to track open devices. These functions allow the user space utilities to build a tree of dependencies between the device mapper targets. The device mapper checks if a particular target closed all its devices after calling its destructor to aid finding programming errors. When adding a new snapshot device, all existing views must open this new snapshot device.

Given the constraints of the device mapper API, the `dm_origin` and `dm_cow` structures had to be expanded to hold a fixed amount of pointers to open devices and target structures. This does not limit the number of snapshots that can be created, but it does limit the number of views that can be created for one origin device.

Map Operation

The `map` method of the `esnapshot-org` is the heart of the whole system. All *read* operations are remapped to the underlying *origin* disk. When a *write* operation is intercepted, the following things are checked before passing the operation to the *origin* disk:

1. If the chunk was already copied, pass the operation to the *origin* device.
2. If there is a copy-on-write operation in progress on the given chunk, the new I/O operation is added to the queue of waiting I/O operations belonging to this chunk. When the copy-on-write operation is completed, this I/O operation is sent to the underlying device.
3. Else start the copy-on-write operation.

The copy-on-write operation is assigned next free chunk from the snapshot area. It is then passed to the `kcopyd` daemon that handles the copying and when it is finished, it calls an appropriate call-back function.

4.2.2 Target esnapshot

The target *esnapshot* represents a particular snapshot. It is not exactly clear what this particular target should be used for. From the user point of view and from the point of integrating the new snapshot architecture into the LVM it is better when every allocated logical volume (i.e. the COW device) has a real existing device node in the file-system. Theoretically, the *esnapshot* target could behave as a read-only device that would present the snapshot of the *origin* device.

However, there are some arguments against such behavior. First, most of the snapshots are accessed rather rarely, at the same time the *esnapshot* target is created for every snapshot. As only the tables of the latest snapshot are required to be in memory if the user does not access any view, making the *esnapshot* target serve the snapshot data would force all exception tables to be in memory all the time. On the other hand, if the *esnapshot* target did not do anything, the snapshot tables would be loaded only if the user requested access to old data through the *esnapshot-view* target.

It is also problematic to indicate to other processes accessing the target that the device is supposed to be read-only. Without such indication the filesystem expects the device to be writable and surprising behavior may occur. The only remaining function of the *esnapshot* target is that it can be monitored by *dmeventd* to detect when the snapshot is getting full.

Therefore it was decided that the *esnapshot* target would not serve any I/O operations and would be provided only as a node that is sent an ‘event’ when an I/O operation on the *esnapshot-org* target cannot be completed because the COW area is full.

The *esnapshot* device is used to send a message to the *esnapshot-org* target to notify that the snapshot area was extended. The LVM automatically suspends and resumes the extended COW device, the `resume` method of the *esnapshot* target checks

for a change of size of the underlying device. The *esnapshot-org* target need not be suspended and performance is thus slightly improved (the underlying device holding snapshot data must be suspended anyway).

4.2.3 Target esnapshot-view

The esnapshot-view target presents a snapshot of the *origin* device to the user. The volume can be written on; all modifications are saved in a separate view COW area and do not destroy the snapshot from which the view is derived.

The target command line has following format:

```
... view_device origin_device snapshot_device
```

The constructor of the view target finds an appropriate `dm_origin` structure by searching a linked list of all `dm_origin` structures and comparing the view device. Similarly it finds an appropriate `dm_cow` structure for the snapshot device and loads the exception tables from the disk if they are not loaded yet.

A new `dm_cow` structure is then created to accommodate the write operations directed on the view. When the view gets full, the event is sent to the dmeventd process listening to the view target.

4.2.4 Implementation of the disk journal

Once the basic snapshot framework is implemented, it could be extended with a journal functionality to additionally improve performance. The changes to the snapshot code include only changes in the `map` function of the origin target; the `map` function of the view target is not affected.

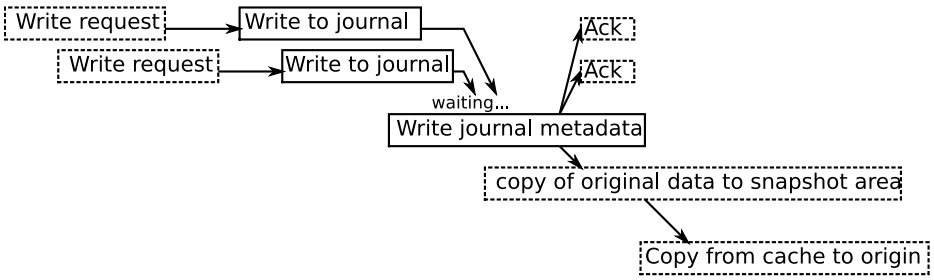


Figure 4.11: Disk caching data flow overview

There is a high degree of serialization in the journal framework; this is partly to save the space, partly to reduce complexity of handling out-of-sequence updates of metadata structures. Considering that most of new disk systems contain large write caches, it should not cause significant performance reduction. The disk structure contains

header, data and meta-data blocks. The meta-data blocks describing the location of written data directly succeed the corresponding data blocks.

When a *write* operation is requested, the system first detects whether there is enough space in the COW device. If there is, a structure `jentry` describing the block in the journal is allocated. The `jentry` structure is connected to the `pending_exception`, so that the *cache writer* can be notified when the copy-on-write operation is completed. A kernel thread *meta-data writer* is then started; the *meta-data writer* waits for a completion of blocks exactly in the order how they are being written in the journal. When a meta-data segment is full or when there are no more write requests, the meta-data block is written to the disk. Once the meta-data is safely written, the I/O operations are acknowledged, the `jentry` structure is linked into a `read_hash` table and the `jentry` structure is sent to a *cache writer* kernel thread.

The *cache writer* thread processes the `jentry` items in a FIFO queue. The thread first waits until the copy-on-write operation concerning the `jentry` is completed and a write can proceed. The `kcopyd` thread is then instructed to copy the appropriate data from journal to the origin disk. When a duplicate write is detected, the *cache writer* waits until the duplicate operation is completed.

The *read* operation without disk journal is handled by directing the read to the origin disk. When disk journal is used, hash table of pending `jentry` structures is searched in order to find a write operation to the same chunk. If one is found, the read operation is chained to the most recent `jentry`, so that it gets executed when the data gets copied from the journal to the origin disk by the *cache writer*. It is not expected that this code path will be taken often as the system caches all writes and reads and most of the reads will be actually performed from the memory buffers bypassing the actual disk device completely.

The journal device works in a cyclical mode — when the end of the device is reached, the data starts being overwritten from the beginning. In order to minimize disk accesses, the oldest valid block of the journal is stored in the journal header. When valid data needs to be rewritten because the end of the journal was reached, the pointer to the oldest valid block is simply moved forward and the header is saved. This operation evades rewriting metadata to make new space available for incoming data, a lot of space can be released for the new data only by changing the header.

4.2.5 Error Handling

The error handling is a very important part of every disk target. Errors do occur and the snapshot device should act in order to reduce impact on the production data when it happens.

When an error occurs on the COW device, the COW device is disabled and the operations to the *origin* device can proceed. The COW device is marked as invalid, so that it cannot be subsequently used. However, it is possible that the write operation modifying the COW header will not reach the disk.

When an error occurs on the journal device, the journal is flushed and the *origin* disk is marked as inaccessible. This is to try to ensure that as little corruption as possible occurs. There seems to be low probability that only a part of the data would be affected, HW failure would probably impact all data including *origin*, COW and journal areas.

Slightly more probable complication can occur during memory allocation. *Mempools* are used for most temporary data, in the worst case only the performance should suffer, but failure or deadlock should be avoided. The only remaining place is an `exception` structure used to remind the redirections on the last COW device. The allocation of this structure may fail. In such case the COW device is invalidated, which invalidates the whole snapshot chain. Not much can be done to help in such situation, perhaps some information in the journal could help to resume normal operation when the memory is available again.

4.3 Device Mapper Userspace Library

The *libdevmapper* library is a userspace part of the device mapper; it provides methods for creating new device mapper devices, loading tables, suspending, resuming, fetching information; the *dmeventd* daemon is part of the *lidevmapper* library. The *libdevmapper* library was written to explicitly support applications similar to LVM. It allows the application to build a tree-like structure describing the device mapper device and their dependencies.

The library was extended with functions `dm_tree_node_add_esnapshot_target`, `dm_tree_node_add_esnapshot_target_snap` and `dm_tree_node_add_esnapshot_target_view` to facilitate adding *esnapshot-org*, *esnapshot* and *esnapshot-view* targets respectively.

The *esnashot-org* requires specification of multiple snapshot devices. The *libdevmapper* library already supports some targets that require multiple devices, such as striped or linear targets — the function `dm_tree_add_target_area` is used for such arguments.

4.4 Logical Volume Manager

As the Logical Volume Manager is used to manage original Linux snapshots, it was natural to extend it to support the new snapshot/view framework. The old snapshot driver provided snapshots that were independent and writable — essentially a combination of snapshot and view.

The new snapshot driver requires the LVM framework to manage a chain of snapshots and the snapshot/view relationship. A new segment type, *esnapshot* was created which links the origin disk with appropriate snapshot areas. There are not many ways how to represent a snapshot in LVM. The COW area is a piece of allocated disk space which is naturally represented by a logical volume in LVM. Logical volume should

have a device in the volume group directory, therefore the kernel driver should support a device to represent the snapshot. This representation allows the users to assign symbolic names to the snapshots.

Information needed to properly manage the view is very similar to the old snapshot — the snapshot source of the view and the view snapshot area. Thus, same approach is taken — new segment type *esnapview* was added that contains links the view disk area to the correct snapshot.

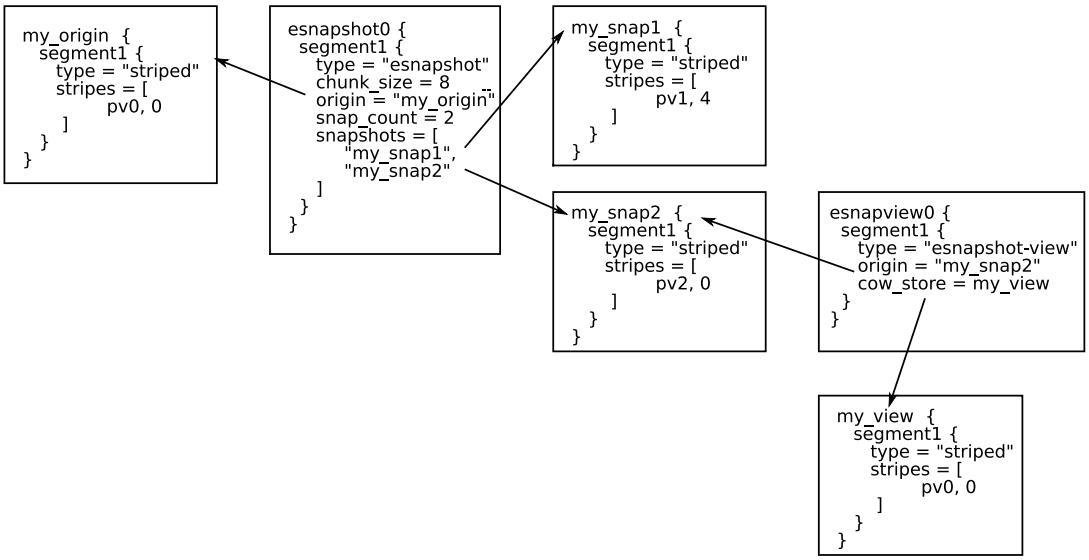


Figure 4.12: Logical volume manager internal structures

The logical volume manager supports creating, removing and extending snapshots and views and displaying information about the use of the snapshot and view areas. The commands `lvcreate`, `lvremove`, `lvextend` and `lvdisplay` had to be extended.

The `lvcreate`, `lvremove` and `lvextend` operations generally first suspend the device, perform the requested operation and then resume the device. The aim should be to limit the impact of the operations on the devices.

When a new snapshot is created, the origin device is suspended and filesystem lock is requested. This operations should flush all pending filesystem buffers and put filesystem into consistent state; it ensures that the data that was supposed to be on the disk in time of snapshot request does not get lost somewhere in system buffers.

The `lvremove` operation requires *noflush* version of suspend, so that the snapshot can be removed even when I/O operations are queued waiting for a COW area to be extended. There is a slight change when removing a view; when a view is being removed, the normal suspend is performed. However, the view detects that normal suspend was called and sends all waiting I/O operations an error message. Otherwise a deadlock would occur.

The `lvextend` operation does not impact the origin device when a snapshot is being extended. Instead of the dumb snapshot device is suspended and resumed. *Noflush* suspend is used when extending a view.

It may happen that at the same time a new snapshot is being created, the I/O that is being flushed requires the last COW device to be extended. Unfortunately, since the moment the LVM starts suspending the *origin* disk all LVM structures get locked and no process can change on-disk configuration of the LVM volumes; the old COW device cannot be extended. This limitation is to allow the reconfiguration of system disks while avoiding a deadlock — when a device is suspended, even the LVM may not touch the disk. This limitation had to be lifted, so that the last COW area could be extended even when a suspend operation is in progress. Therefore, the new snapshot driver is unsuitable for filesystem containing LVM configuration information (usually the `/etc` filesystem).

4.5 Device Monitoring and Auto-extension

The auto-extending snapshot and view areas are an important feature to ease administration. The *dmeventd* daemon is provided by the libdevmapper library to allow monitoring of device mapper devices. The LVM is extended to build a library that is later registered with *dmeventd* to monitor certain devices.

The code that defines the LVM segments handles the registration part. The *dmeventd* is asked to monitor the latest snapshot dumb device and all view devices. The code to handle both snapshot and view devices is the same. When it receives an event, it asks the device whether it is full. If it is, the `lvextend` command is issued on the device.

4.6 Summary

New snapshot driver was proposed with the following properties: the performance should be independent on the number of snapshots, snapshot/view model should be implemented and automatic on-demand snapshot area exapnsion should be available. The new snapshot driver should be integrated into some administrative environment.

All these goals were met, however the Logical Volume Manager, which was used as an administrative environment, is not friendly to new device types. The changes to the code are scattered through the whole code base and unfortunately not systematic. Architecture change in LVM would be needed to better accomodate these new device types.

Additionally, a journalled device was implemented that was supposed to cache execution of the write operations and possibly in the future provide any-point-in-time restores. Unfortunately, some problems appeared during testing the performance of the journalled device, further development is needed to find and identify the reason for the differences in the performance.

Chapter 5

Benchmarks

The tests were run on an Intel Xeon Quad Core server with 4 GB of main memory. As a storage a Promise VTRak E610s disk subsystem with Serial SCSI disks configured to use RAID5 redundancy and 512MB of write cache. The storage was connected using 2 GB Fibre Channel SAN connection.

The performance was first measured using UNIX `dd` command, however such method does not properly reflect real world disk usage. Next an *IOmeter* [11] tool formerly developed by Intel was used, but the test results were not consistent with the test settings of the tests. In the end, the *fio* tool [13] was chosen and proved to be reasonably easy to configure and provided consistent results.

As the primary goal of this thesis was to improve write performance, the tests were configured to be only write tests. The *fio* tool allows the user to specify different block sizes. A 4K, 128K and a mix emulating *IOmeter* file server access pattern were used. The number of concurrent I/O operations was left on the default values, 64. Other values were tested as well, higher numbers can generally saturate the disks better, the lower numbers simulate better that many operations even on highly loaded systems have to be serialized. The differences between different concurrent I/O were not significant.

The values that are typically observed are throughput and number of I/O operations per second. The first value is typically important for data intensive application, the second is important for transaction systems. It should be noted that the larger the data block, the smaller becomes the number of I/O operations, as each operation writes more data. However, this changes when using small data blocks as the work the system has to do with e.g. 512 byte write operation is often roughly the same as with 64 KB write operation. The copy-on-write operation of the snapshots has to copy at least one chunk for every write operation regardless how small it is.

In the following text only the throughput values are explained. The I/O per second values are generally proportional to the throughput and indirectly proportional to the block size. The performance of the Promise disk system as measured by the *fio* program is shown in table 5.1.

	4 KB block writes	64 KB block writes	mixed blocks
Throughput	28,459 KB/s	154,000 KB/s	61,000 KB/s

Table 5.1: Raw disk performance

It should be noted that the `dd` program achieved write performance of about 200 MB/s.

5.1 Number of Snapshots

The number of snapshots very significantly lowers the performance of the old snapshot driver, while the new snapshot driver is totally unaffected. The effect is very significant on the tests with 4K and 64K blocks. After the tests were run it became obvious that having more than 10 snapshots on 1 disk renders the disk useless.

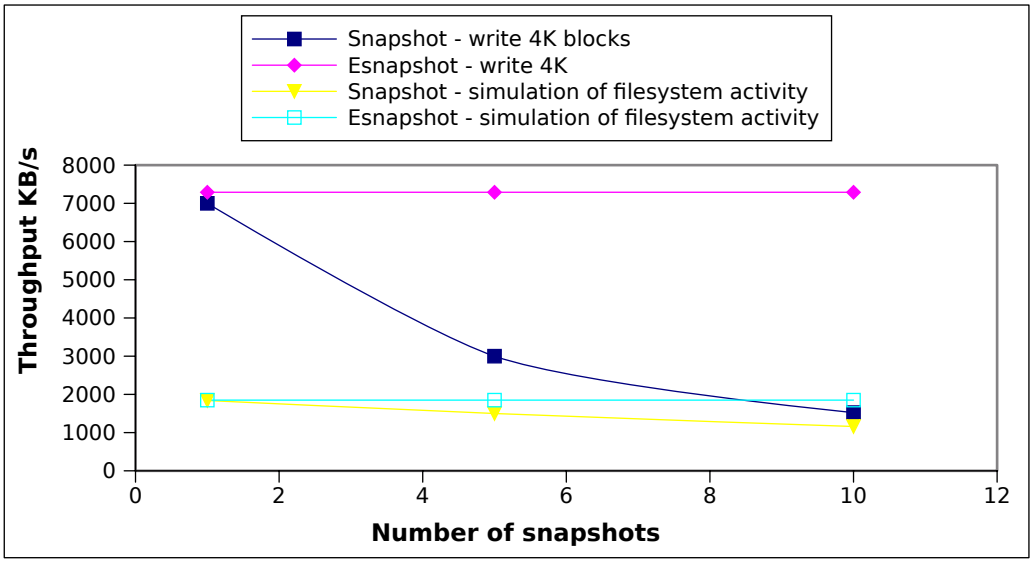


Figure 5.1: Impact of multiple snapshots on the performance

The architecture of the new snapshot driver makes the performance independent on the number of snapshots, but when using 1 snapshots, the design is practically identical which results in similar performance. The differences can be attributed either to slightly better code optimization or some randomness in the testing system.

5.2 Test Data Block Size and Chunk Size

The larger the block size, the faster the disk subsystem can serve the data. One 128KB block will be served faster then 32 4KB blocks from different locations. The same applies to write operations as the RAID5 system needs to read some data to correctly compute the redundant parity, therefore larger writes are more efficient.

If a chunk size is larger then the data block size, there is some probability that the testing process will try to repeat a write to a chunk, that was already copied to the COW area; the write operation thus proceeds directly to the disk and speeds up the execution.

Combination of these 2 factors leads to an interesting result: the tests with large blocks are fastest when used with small chunk size, the tests with small blocks are faster when used with large chunk size.

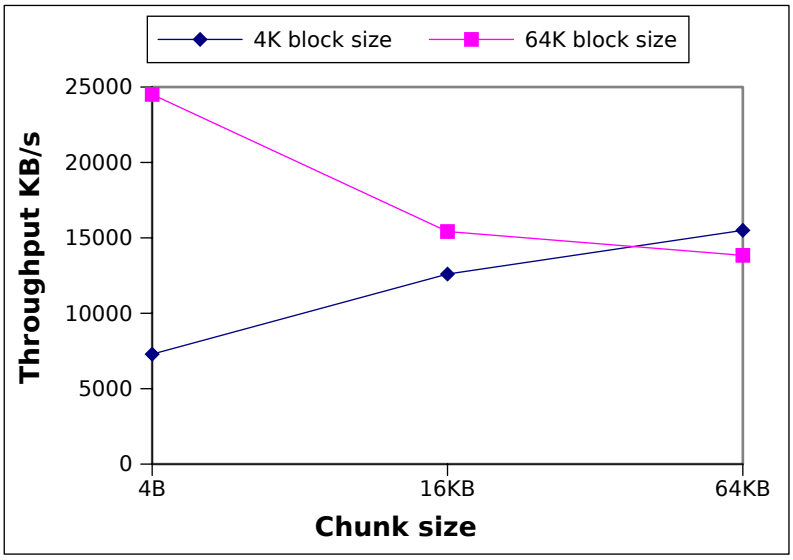


Figure 5.2: Impact of chunk size and block size

5.3 Journal Performance

I measured the journal performance using the same methods as I did the normal snapshots. Although the results using the `dd` program seemed very promising, both *IOmeter* and *fio* reported abysmal numbers of the magnitude of 0.5 MB/s.

A lot of tests were performed including some changes in the architecture and the bottleneck was not found. This is especially peculiar as the architecture of journal is independent on the order of the writes; the performance should be roughly identical using either of the sequential and random access methods. More research should be done to find out the bottleneck in the data flow.

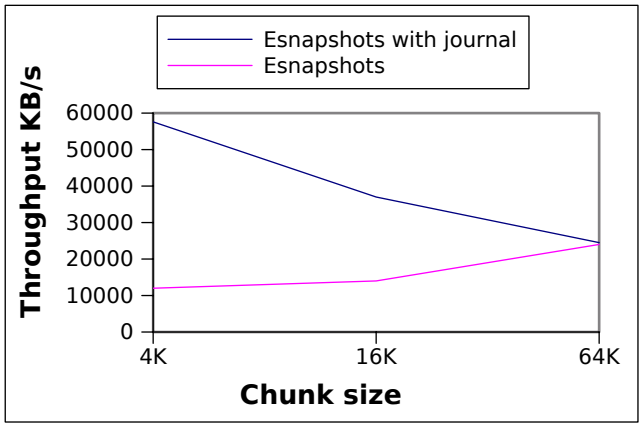


Figure 5.3: Throughput measured by the program `dd`

It should be noted that the journal can only serve as a temporary buffer for higher write activity; the journal needs to be flushed into the *origin* disk anyway and the copy operation itself adds more processing when reading the data from the journal. Unfortunately, the data written to the journal cannot be in current kernel architecture kept in memory and have to be read from the disk again.

Chapter 6

Conclusion

6.1 Achievements

This thesis aimed to develop a time accessible storage device for Linux, in particular a new snapshot kernel virtual device that could be used for example in a virtualization appliance to provide backup services.

The developed snapshot driver accomplishes the aim; it provides snapshot/view architecture common in storage appliances; it uses chained snapshot to achieve a situation where additional snapshots do not degrade the performance further. The solution was completely integrated into Logical Volume Manager framework to achieve simpler administration. Automatic snapshot extension was added, so that the disk space can be allocated more efficiently. Additional disk journal was proposed as a means to improve performance of the solution, with the future possible use to provide any-point-in-time recovery.

The copy-on-write snapshots have very significant performance impact on the performance of the production storage. As described in chapter 5, the throughput of the system can easily degrade more then 60%, more if non-chained snapshots are used. The introduction of chained snapshots helped to keep this degradation stable regardless of the number of snapshots.

Disk journal was proposed and implemented to reduce the performance impact of the snapshot solution. Unfortunately, confusing results were obtained from the tests; the speed improvement was visible only on certain type of access. More work should be targeted at finding the bottleneck in the data flow, however even then it is very likely that the disk journal would be used to add any-point-in-time recovery functionality instead of using it to gain more performance.

6.2 Future Work

The new snapshot driver consists of two distinct parts. The snapshot logic handles copy-on-write operations and a write journal is used to improve the performance.

Functionality of both parts can be extended in several different areas.

The journal disk cache could be additionally extended to contain time marks and information about snapshots. Full Continuous Data Protection device with unlimited time granularity can be facilitated by combining snapshots with journal data.

Both the snapshot data and journal could be used to provide data for some form of delayed asynchronous replication. This data can be either sent to a remote location or saved on a backup media and serve as a block-level incremental backup. New interfaces would have to be devised to provide access to the required data.

The solutions in the device mapper layer cannot easily use memory buffers to avoid some unnecessary reading data from the disks. Moving the snapshot logic into different part of the operating system should be considered with regard to intended use. More choices are available if the snapshot services are supposed to be exported through iSCSI or FC target technologies.

Safe storage of data is of paramount importance in current business environment. This includes both the ability to safely preserve old versions data and confidence that the storage solution does not corrupt the data in the process. Good design, implementation and rigorous testing are necessary in addition to software maintenance and support. The actual implementation is only a part of work necessary to make this solution part of the real-world production environment.

Bibliography

[1] Linux kernel sources, http://www.kernel.org.

[2] Peterson Z. N. J., Burns R.: Ext3cow: A time-shifting File System for Regulatory Compliance, http://znjp.com/papers/peterson-tos05.pdf

[3] Dwivedi H., Securing Storage: A Practical Guide to SAN and NAS Security, Addison-Wesley Publishing, 2005, http://media.techtarget.com/searchStorageChannel/downloads/Dwivedi_ch02.pdf

[4] Reisner P.: DRBD, 2000, http://www.drbd.org/fileadmin/drbd/publications/drbd_paper_for_LK7.ps.gz

[5] Causes of Downtime and Cost/Operational Effects, Libelle corporation, http://www.libelle.com/en/index.php?option=com_docman&task=doc_download&gid=36

[6] Freeman L., Looking Beyond Hype: Evaluating Data Deduplication, Network Appliance Inc. White Paper, 2007, http://www-download.netapp.com/edm/TT/docs/Looking_beyond_hype_Dedupe.pdf

[7] List of backup targets with deduplication, http://www.backupcentral.com/components/com_mambowiki/index.php/Disk_Targets%2C_currently_shipping

[8] Fujita T., Christie M.: tgt: Framework for Storage Target Drivers, 2006, http://www.kernel.org/doc/ols/2006/ols2006v1-pages-303-312.pdf

[9] Fujita T., Ogawara M.: Analysis of iSCSI Target Software, http://zaal.org/paper/snapi2004.pdf

[10] Petr Dvorak: Business Continuity, master thesis VSE Praha, 2004

[11] Iometer — I/O subsystem measurement and characterization tool for single and clustered systems, http://www.iometer.org

[12] Hitachi TrueCopy Remote Replication, http://www.hds.com/products/storage-software/truecopy-remote-replication.html

[13] I/O tool for benchmark and stress/hardware verification, http://freshmeat.net/projects/fio/

[14] R1Soft CDP Solutions, http://www.r1soft.com/CDP.html

Appendix A

Contents of the Enclosed CD

The enclosed CD contains the following items:

- The file `thesis.pdf` contains the text of this thesis.
- The directory `esnap` contains the source code of the new kernel snapshot driver.
- The directory `device-mapper` contains source code of the *libdevmapper* library with added changes to support new snapshots.
- The directory LVM2 contains source code of the Linux Volume Manager with added changes to support new snapshots.
- The files `lvm.diff` and `device-mapper.diff` contain patches that can be applied on the CVS checkout of the LVM2 and device mapper respectively. The particular dates of the checkout are described later in this appendix.

Appendix B

Compilation and Installation

There are 3 subsystems that must be compiled and installed in order to run the new snapshot driver.

B.1 Kernel driver

The kernel driver resides in the directory `esnap` on the installation CD. The `Makefile` should be changed so that the `KERNEL_SRC` variable points to the configured source tree of the running Linux kernel, or, alternatively to the `/lib/modules/<kversion>/build` directory of the running kernel. The command `make` then builds the snapshot kernel module `dm-esnapshot.ko`.

The compilation was tested on kernel 2.6.26 on the i386 platform. Although care was taken to write the code without reference to any particular platform, it was not tested on any 64-bit platform and it is therefore likely that some problems may arise.

There were some changes in the file names of the device mapper files recently, the module will not compile on older kernel versions. The kernel must be compiled with the support of device mapper architecture (the module `dm_mod.ko`).

When the module is compiled, it should be installed into the directory `/lib/modules/<kversion>/kernel/drivers/md/` and the command `depmod -a` should be run to update the `modprobe` tables. This ensures that LVM automatically finds and loads the module.

B.2 Libdevmapper

The installation CD contains a complete source code in the directory device-mapper, as well as a diff file against CVS checkout dated Jun 24 18:16. The following commands will compile and install the libdevmapper:

```
$ ./configure --enable-dmeventd
$ make
$ make install
```

B.3 Logical Volume Manager

The installation CD contains a complete source code in the LVM2 subdirectory and a diff file against CVS checkout date June 24 17:55. The following commands compile and install the LVM:

```
$ ./configure --enable-cmdlib --enable-dmeventd
$ make
$ make install
```

The user must take care that no old versions of `liblvm2cmd.so` library stay in place. The *dmeventd* daemon would not work correctly if linked against this library.

In order to configure automatic expansion of COW areas, the esnapshot libdevmapper library must be enabled in the `/etc/lvm/lvm.conf` configuration file in section dmeventd:

```
dmeventd {
  esnapshot_library = "libdevmapper-event-lvm2esnapshot.so"
}
```

The amount of space that should be appended automatically to a full snapshot area can be configured by a parameter `autoextend_extents` in the global section. The parameter contains the number of extents (the smallest chunk of data that is allocatable to a logical volume). If a lot of traffic is expected on the snapshotted volumes, a larger value should be used.

Appendix C

LVM Administration Commands

C.1 lvcreate

The `lvcreate` command is used to create new snapshots, views and for adding a journal to the existing *origin* device.

```
$ lvcreate --esnapshot -L 128M -n snap1
           /dev/testvg/my_origin
```

This commands creates a new snapshot. The preallocated COW area size is 128MB. The name of the snapshot is 'snap1'. Automatic scripts may wish to include a date and time into the snapshot name.

```
$ lvcreate --esnapview -L 128M -n view1 /dev/testvg/snap1
```

This command creates a new view based on the snapshot 'snap1'. The view COW area is allocated 128MB of free space, the view device name would be `/dev/testvg/view1`.

```
$ lvcreate --esnapjrnl -L 1024M -n journal_origin
             /dev/testvg/my_origin
```

This command adds a journal to an existing *origin* device. The size of the journal is 1024MB, the journal is circular.

C.2 lvchange

The `lvchange` command can be used to enable or disable monitoring of a particular device and thus the automatic expansion of the COW area. Monitoring can be affected on the *origin* and view devices. The following commands disable and enable monitoring.

```
$ lvchange --monitor n /dev/testvg/my_origin
$ lvchange --monitor y /dev/testvg/my_origin
```

C.3 lvremove

The `lvremove` command can be used to remove the snapshots, views and journal from the *origin* disk. Only the oldest snapshot can be currently removed. If removing the last remaining snapshot, the journal must be removed first (there is no point in having a journal on top of an origin with no COW areas).

The following command removes a snapshot from an *origin* device.

```
$ lvremove /dev/testvg/snap1
```

C.4 lvextend

The `lvextend` command can be used to extend the *origin*, COW and view logical volumes. The journal device cannot be currently extended, it must be removed and created again bigger. However, such operation does not disrupt the read/write operations directed to the *origin* or view devices.

The following command extends a device by 128MB:

```
$ lvextend -L +128M /dev/testvg/snap1
```

C.5 lvdisplay

The `lvdisplay` command displays information about the volumes. The snapshot and journal devices do not appear in the ordinary list, a parameter `-a` must be added for these devices to appear. The following command displays all logical volumes in a system:

```
$ lvdisplay -a
```